BROWN UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

Thesis

SPECULATIVE COMPILATION OF COMPLEX UDFS IN PYTHON DATA SCIENCE

by

YUNZHI SHAO

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science

2022

Acknowledgments

I would like to thank Leonhard Spiegelberg, the lead developer of Tuplex, and my research advisor Professor Malte Schwarzkopf for their patience, support, and guidance throughout this project.

Contents

1 Introduction **1**

1.1 Problem Statement 1

1.2 Background and Related work 3

1.2.1 Numba 4

1.2.2 Pypy 4

1.2.3 Tuplex 4

1.2.4 LLVM 5

1.3 Contribution 5

2 Language Constructs **6**

2.1 List Comprehensions 7

2.2 Loops 7

2.3 Iterators 9

3 Design **13**

3.1 Type Stability 13

3.2 Solution 15

4 Implementation **17**

4.1 Overview 17

4.2 Loops 18

4.3 Iterators 19

4.4 Loop Unrolling 23
4.4.1 Full Loop Unrolling 24
4.4.2 Partial Loop Unrolling 25
4.5 Type Tracing 26

5 Evaluation 28

5.1 Setup 28
5.2 Experiment I: Numeric-heavy workload 28
5.3 Experiment II: Experiment where Numba is better 32
5.4 Numba Limitation 33

6 Conclusions and Future Work 35

References 36

Chapter 1

Introduction

1.1 Problem Statement

Python has been the most popular language for data science for its simple, flexible syntax and wide variety of libraries available. Many Python data science pipelines involve UDFs (user-defined functions) [1], allowing for complex and customized computation to be performed over datasets. However, running Python UDFs through the native CPython interpreter can be slow. Among many attempts to speed up Python, Tuplex [1], a new data analytics framework developed at Brown, showed that despite the challenges to run Python efficiently, there is a potential to speed up data science using speculative compilation techniques. To achieve this, Tuplex introduces *data-driven compilation* and *dual-case processing* to JIT (just-in-time) compile Python UDFs into optimized machine code through LLVM [1].

Python is a dynamically typed language, indicating that a variable's type is usually unknown until runtime and can get reassigned anytime during execution. The highly dynamic nature of Python code makes it challenging to compile Python UDFs to efficient machine code. Although users can provide type hints to attach type annotations to variables [2], it is optional and not (yet) required or enforced by the language itself. As explicitly annotating code hampers productivity, users often prefer to not use type annotations in practice. While it's often difficult or even impossible to deduce types by only looking at the source code, for Python UDFs within a data ana-

lytics pipelines it is possible. As such UDFs can leverage the implicit query structure, this opens up the possibility of UDF compilation without explicit type annotations [1]. In addition to speculative compilation, Tuplex establishes a dual-case processing model that separates input rows into a common case and an exceptional case [1]. Tuplex focuses on compiling the common case only by inferring and taking advantage of expected common input types to allow for more optimization while handling rarer exceptional cases through a fallback path (i.e., the CPython interpreter).

Prior to the work presented in this thesis, Tuplex did not have support for compilation of loops. However, loops are one of the most fundamental and powerful statements in almost every programming language, including Python. Python UDFs in big data queries often unavoidably rely on loops as part of the computation. Bringing loop support to UDF is therefore a requirement to make it more useful in the real-world, as loops are prevalent in real-world applications. To demonstrate this, confer a snippet (Listing 1) from a Kaggle notebook [3] showing a UDF using 5 nested loops to calculate traffic density from an input data frame.

Tuplex has shown that compiling UDFs helps it to outperform general-purpose Python compilers by 6-24x, and is the critical factor that it outperforms other data analytics frameworks like Spark and Dask [1]. We have reason to assume that adding compilation support for loops would result in a similar speedup.

However, compiling Python loop statements to LLVM IR (subsection 1.2.4) can be challenging. First, the for statement has a relatively flexible syntax that allows iterating over differently typed objects (e.g., list, tuple, string, range, iterator). Second, loops can have multiple statements that change the control flow: `if`, `continue` and `break`. Third, loops can be nested. Finally, a variable inside a loop can change its type over iterations, introducing significant complexity in generating statically typed LLVM IR.

```
def calculate_traffic_density(df):
    # some lines are omitted ...

    for i in range(n_bins_lon+1):
        bins_lon[i] = BB_traffic[0] + i * delta_lon
    for j in range(n_bins_lat+1):
        bins_lat[j] = BB_traffic[2] + j * delta_lat

    for y in range(n_years):
        for d in range(n_weekdays):
            for h in range(n_hours):
                idx = (df.year==(2009+y)) & (df.weekday==d) &
                ↪  (df.hour==h)
                inds_pickup_lon =
                ↪  np.digitize(df[idx].pickup_longitude,  bins_lon)
                inds_pickup_lat = np.digitize(df[idx].pickup_latitude,
                ↪  bins_lat)

                for i in range(n_bins_lon):
                    for j in range(n_bins_lat):
                        traffic[y, d, h, j, i] = traffic[y, d, h, j,
                        ↪  i] + np.sum((inds_pickup_lon==i+1) &
                        ↪  (inds_pickup_lat==j+1))

    return traffic
```

Listing 1: Example of a UDF in data science that contains loops

This thesis presents three new features we have brought into Tuplex: the support for compiling loops, the support for using a subset of iterators, and internal type tracing for loops. We show the difficulty involved in the implementation of loops and iterators and explain our solution. In the end, we illustrate the improvement in Tuplex's performance achieved by our work.

1.2 Background and Related work

1.2.1 Numba

Numba [4] is a JIT compiler that compiles a subset of Python code — including functions from NumPy — into machine code using LLVM. It has optimization on loops including vectorization, and is generally much faster than the CPython interpreter. It focuses on compiling numeric UDFs and does not support certain Python features (e.g., `reversed()`, `next()` with two arguments). In addition, it makes strict assumptions for certain data types, e.g., lists supported in Numba must be homogeneous (i.e., have elements of the same type) [1] [5]. Numba is mainly used to speed up numerically-oriented Python functions.

1.2.2 Pypy

Pypy [6] aims to serve as a faster alternative as CPython. It uses a tracing JIT compiler to achieve optimization by detecting frequently executed loops and generating highly optimized machine code for only the commonly executed code paths [7]. Compared to Numba, Pypy focuses on providing more comprehensive support for language features, but has less optimization in terms of individual UDFs.

1.2.3 Tuplex

Tuplex's Python UDF compiler is embedded as part of its map operator [1]. Tuplex uses a dual-mode processing model and compiles Python UDFs to machine code through LLVM for the common case detected. For exceptional rows that do not agree with the common case or uncompilable UDFs, Tuplex uses the Python interpreter as a fallback path. Prior to our implementation, Tuplex did not support the compilation of loops and would need to run UDFs with loops through the Python interpreter.

[1] Tuplex also requires the type for lists to be homogeneous to allow more optimization.

1.2.4 LLVM

LLVM is a compiler framework with JIT compilation support widely used in compiler development for its portability and high performance. It provides language-independent optimization and cross-platform code generation support [8]. The LLVM libraries are built around the code representation called LLVM IR (intermediate representation). In LLVM, local variables are declared either in registers or in memory (on stack) [9]. Variables in registers need to follow SSA (static single assignment form), which requires that a variable can only be assigned once.

1.3 Contribution

We implemented compilation support for loops, including the `for` loop and the `while` loop. To allow more comprehensive support for loops, we improved the list implementation to support list of tuples and added support for built-in iterator-related functions. Although some simple loops can be compiled straightforwardly, generating efficient LLVM IR code for general Python loops is challenging, as a Python variable's type can be inconsistent over iterations. We addressed problems related to type stability through loop unrolling and traced typing.

As a result, Tuplex can compile UDFs with loops and a subset of iterators without having to switch to the fallback path and use the single-threaded Python interpreter. This speeds up the execution by around 26x for single-threaded Tuplex and up to 85x for 4-way parallel Tuplex [2].

[2] The multi-threaded experiment was run with `executorCount = 3` set in Tuplex context configuration. Both speedups are computed by comparing with the single-threaded CPython interpreter, which is used in fallback mode. The benchmark experiment can be found in Chapter 5.

Chapter 2

Language Constructs

In Python, there are three different ways to implement loops:

- **Comprehensions:** Comprehensions are fast and convenient ways to create lists (can also be used to create tuples, sets or dictionaries) from existing iterable objects. Although comprehensions use the `for` keyword and are internally implemented as loops, their usage is limited.

- **For loops:** A `for` loop is constructed using the `for` keyword followed by one or more target loop variables, one or more expressions to be iterated over, one or more statements to be repeatedly executed, and an optional `else` clause. `for` loops are typically used to iterate over a sequence (e.g., list, tuple, string and range object) and perform operations on each element within the sequence.

- **while loops:** A `while` loop is constructed using the `while` keyword followed by an expression that yields a boolean value, one or more statements, and an optional `else` clause. `while` loops are used when we need to repeatedly execute some statements providing that the expression yields `true`.

Each of these language constructs have their own unique challenges when it comes to generating LLVM IR semantically-equivalent to Python's interpreted execution. In the following sections, we illustrate our approach for adding compilation of loop constructs to Tuplex.

2.1 List Comprehensions

List comprehensions provide a convenient but limited way to create lists using a bracket with an expression, a for clause and optional if clauses [10]. A `for` loop that finds integers within $[0, 100)$ not divisible by 7, as follows:

```
1  lst = []
2  for i in range(100):
3      if i % 7 != 0:
4          lst.append(i)
```

can be simplified to:

`lst = [i for i in range(100) if i % 7 != 0]`.

Compared to a loop, a list comprehension is more syntactically constrained and simpler to implement. One difference is that a loop overrides the loop control variable (`i` in the loop above) in the surrounding scope, allowing us to retain the value of `i` after the loop. This is not the case in a list comprehension since Python 3 where the `i` is merely a temporary variable within the scope of the list comprehension [11]. In addition, a list comprehension only allows expressions. As a result, it does not contain variable assignments (except for using assignment expressions since Python 3.8 to assign to temporary variables [12]) and does not support the `break`, `continue`, or `return` keywords that complicate the control flow of more complex loops.

2.2 Loops

For iterating over sequence types (e.g., tuple, list, range, string) or other iterable types (e.g., dict, iterator) for more general purposes, a `for` loop is used. A `for` loop

has more flexible and complicated syntax than a list comprehension. The Python language reference [13] defines the `for` statement as:

```
for_stmt ::=  "for" target_list "in" expression_list ":" suite
              ["else" ":" suite]
```

In Tuplex, `expression_list` can be a list, a tuple, multiple comma-separated lists/tuples, a string, or a range function. `target_list` is a variable or multiple variables to which `expression_list` can unpack and assign accordingly. Examples of some newly supported syntax in Tuplex as a result of our work are shown in table 2.1.

`expression_list` type	Supported `for` loop syntax
list	`for a, b in [(1, 2), (3, 4)]: suite`
tuple	`for i in (True, "abcd", 1.0, (1, 2)): suite`
comma-separated lists/tuples	`for a, b in (1, 2), (3, 4), [5, 6]: suite`
string	`for c in "hello, Tuplex": suite`
range	`for i in range(-10, 10, 2): suite`

Table 2.1: Examples of new supported `for` loop syntax

`suite` is a single same-line statement or an indented block of statements. The first `suite` of a `for` loop will be executed once for each item from `expression_list`. The `for` loop also has an optional `else` clause followed by a second `suite`, which will be executed if the loop is not terminated during the first `suite`.

Two of the most interesting statements in the context of loops are `break` and `continue`. They are only allowed in the first `suite` of the nearest enclosing loop. `break` terminates the nearest enclosing loop, and `continue` passes control to the start of the next iteration of the nearest enclosing loop [13]. `break` and `continue` are usually used in conjunction with `if` statements to switch control flow based on

different conditions.

The while statement is also supported in Tuplex. Following the Python language reference [14], a while statement has the syntax:

```
while_stmt ::=  "while" assignment_expression ":" suite
                ["else" ":" suite]
```

where `assignment_expression` evaluates to a boolean value. The implementation of the `while` statement is very similar to that of the `for` statement, except that in a `for` loop, we often can determine the total number of iterations before starting (e.g., from the length of the iterable), but for a `while` loop, the number of iterations is usually not known until `assignment_expression` returns `false` and the loop finishes.

With our work, Tuplex supports `else`, `break` and `continue` for loops in UDFs. It can also handle nested loops and loops containing or under `if` clauses. Those features enhance flexibility and expressiveness for UDFs, but also makes the code generation more complex.

2.3 Iterators

When data scientists use loops in their UDFs, another language construct to work with are iterators. Iterators are special iterable types, and Python for statement naturally supports using iterators as `expression_list`. An overview of the relationships among iterable, iterator, and generator objects is shown in figure 2·1.

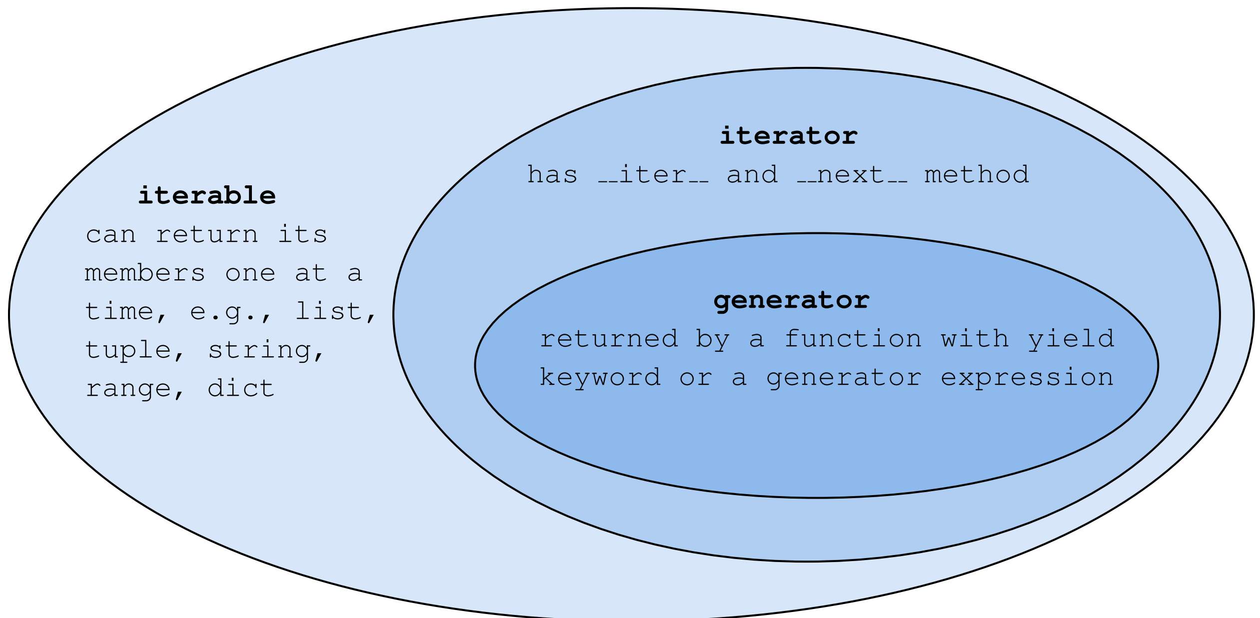

Figure 2·1: iterable, iterator, and generator objects in Python

Conceptually, any object that has implemented the `__iter__` and `__next__` method is considered an iterator [15]. Calling `iter()` on an iterable object returns a corresponding iterator, and calling `iter()` on an iterator simply returns itself.

```
1 lst_iterator = iter([1, 2]) # creates an iterator
2 lst_iterator = iter(lst_iterator) # nothing changes
3 el1 = next(lst_iterator) # el1 = 1
4 el2 = next(lst_iterator, -1) # el2 = 2
5 el3 = next(lst_iterator, -1) # el3 = -1
6 el4 = next(lst_iterator) # exception: StopIteration
```

Listing 2: Example of using iterators to manipulate a list of integers.

We can retrieve the next item from an iterator using the `next(iterator[, default])` call [16] with an optional default value. If no more items are available from the iterator (i.e., iterator is exhausted), calling `next` on it will raise an exception `StopIteration` if no default value is given. Otherwise, the default value will be returned. Listing 2 shows an example of how `iter()` and `next()` work.

In Tuplex, instead of user-defined iterator classes, we primarily focus on Python built-in iterator-related functions that are often used in the context of loops. Specifically, we provide support for iterators generated from built-in functions `iter()`, `reversed()`, `enumerate()`, and `zip()`. Calling `next()` on any returned iterator is also supported. More detailed explanations about those functions can be found in table 2.2, each following the expected behaviors as in the Python Standard Library.

Function	Explanation
`iter(iterable)`	Returns an iterator. `iterable` can be a list, a tuple with homogeneous element types, a string, a range object, or an iterator object.
`reversed(seq)`	Returns an iterator that will return items from `seq` in reversed order. `seq` can be a list, a tuple with homogeneous element types, a string or a range object.
`enumerate(iterable[, start])`	Returns an iterator with that will a tuple consists of an index and the next item from `iterable`. The index starts at `start` for the first item. `start` is 0 by default.
`zip(*iterables)`	Returns an iterator that will return a tuple consists of next items from each `iterable` in `*iterables`. `*iterables` are comma-separated `iterable` objects. The returned iterator becomes exhausted if any `iterable` has run out of items.
`next(iterator[, default])`	Returns the next item from `iterator`. If `iterator` is exhausted, returns `default` if `default` is provided, otherwise raises exception `StopIteration`.

Table 2.2: New supported built-in iterator-related functions

Iterators extend the supported for `for` loop syntax in Tuplex as shown in table 2.3.

`expression_list` type	Supported `for` loop syntax
list_iterator	`for item in iter([1, 2, 3, 4]): suite`
tuple_iterator	`for item in iter((1, 2, 3, 4)): suite`
string_iterator	`for c in iter("hello, tuplex"): suite`
range_iterator	`for i in iter(range(100)): suite`
reversed	`for item in reversed([1, 2, 3, 4]): suite`
enumerate	`for index, s in enumerate(["ab", "cd"]): suite`
zip	`for num, char in zip([1, 2], "ab"): suite`

Table 2.3: Examples of new supported `for` loop syntax with iterators

Chapter 3

Design

3.1 Type Stability

A well-typed loop is a loop where any variable's type can be determined before the loop begins and will remain unchanged throughout the scope of the loop. Such a loop has static types for all variables used in the loop. A famous yet unsolved conjecture known as Collatz conjecture (or 3n+1 conjecture) [17] states that repeatedly applying a given function f to any positive integer n always yields 1 after a finite number of iterations [1]. Now, a user may want to verify Collatz conjecture up to an `upperbound` by using the UDF in Listing 3. This UDF contains a well-typed loop despite of its complexity: We can infer that `upperbound` and `n` are both integers, and `n` remains as an integer in every iteration of the loop.

```
1 def f(upperbound):
2     for n in range(1, upperbound):
3         while (n != 1):
4             if n % 2 == 0:
5                 n = n // 2
6             else:
7                 n = 3*n + 1
8     return True
```

Listing 3: Example of a well-typed loop: the type of `n` is stable.

[1]If n is even, $f(n) = n/2$. If n is odd, $f(n) = 3 * n + 1$.

On the contrary, a loop contains type stability issues if a variable's type changes during the loop. A straightforward example shown in listing 4 is looping over a heterogeneous tuple. Generating this loop to a loop in statically-typed LLVM IR is complex, as the type of the loop variable `i` is changing in each iteration.

```
1 def loop_tuple(t):
2     for i in ([1, 2], "hello", [-1, -2, -3, -4], "a, b"):
3         t += len(i)
4     return t
```

Listing 4: Loop over a heterogeneous tuple

Function `foo` in Listing 5 demonstrates a more complex scenario where type of a previously declared variable is not stable in a loop. `x` is declared as an integer. In the first iteration of the following `for` loop, `x` stays as an integer at line 4, but gets upcast to float at line 5 as a result of a binary operation with a float number. In the future iterations, the type of `x` will remain as float. As a result, `foo` seems to be returning a float number in the end. It turns out that, even we can infer the type for `x` in each line for this example, generating the loop to LLVM IR can still be difficult, because memory region for a variable is initialized with a fixed type.

```
1 def foo(t):
2     x = 10
3     for i in range(t):
4         x = x + i
5         x = x * 1.5
6     return x
```

Listing 5: Type stability makes compilation challenging.

Detecting type stability issues is also a problem. Can we always assume type change occurs and upcast `x` if a line like `x = x * 1.5` exists in a loop? The function `bar` in listing 6 is a counterexample, where `x` should only be upcast if the input

`t > 10`. In fact, even in `foo`, when `t <= 0`, we would need to return an integer (i.e., 10) since the loop will not run.

```
1 def bar(t):
2     x = 10
3     for i in range(t):
4         x = x + i
5         if (x < t):
6             x = x * 1.5
7     return x
```

Listing 6: Detecting type stability issues can also be challenging.

In `foo` and `bar`, independent of whether the type of `x` gets changed, its type will be finalized and remain stable after the first iteration. But there are also loops that contain variables with more frequent type changes. Listing 7 shows a loop where the type of `x` changes in every iteration.

```
1 def unstable_loop(t):
2     x = 10
3     for i in range(t):
4         if i % 2 == 0:
5             x = 10
6         else:
7             x = "hello"
8     return x
```

Listing 7: Variable types in a loop can be unstable in every iteration.

3.2 Solution

To correctly generate codes for loops that contain potential type stability issues, we propose the following solutions:

1. Loop Unrolling:

   (a) Unroll each iteration of a `for` loop whose `expression_list` is a tuple type.

   (b) Unroll the first iteration of a loop, if at least one variable's type changes during the first iteration and every variable's type is stable after the first iteration.

2. Type Tracing: Sample from inputs and trace the type for variables used in a loop during execution to speculate whether a potential type stability issue can be resolved through unrolling.

3. If any currently unsupported Python language features are encountered, or if applying loop unrolling is not able to resolve a type stability issue, the fallback path is used and the UDF will be executed through the Python interpreter.

Chapter 4

Implementation

4.1 Overview

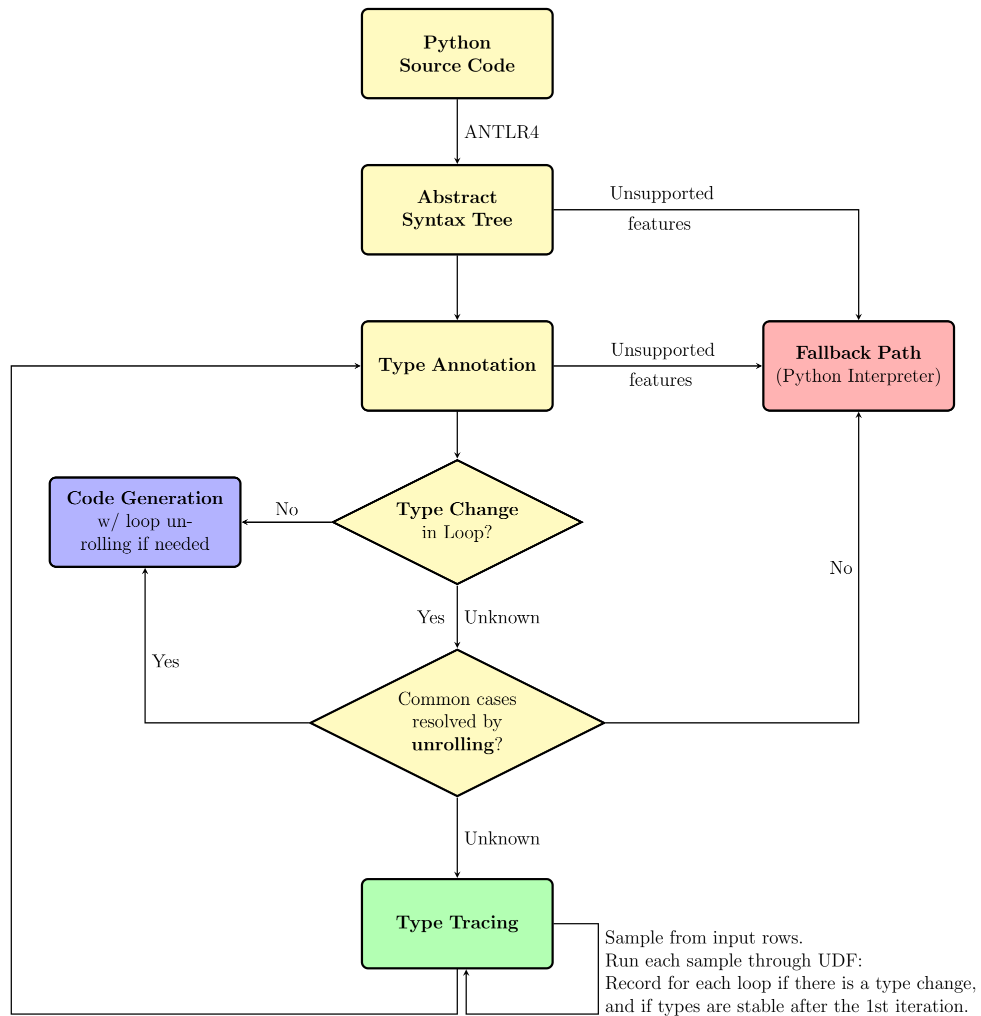

Figure 4·1: Tuplex UDF Compilation Process with Type Tracing

With our work, Python UDFs with loops now can be compiled in Tuplex following the process shown in figure 4·1.

4.2 Loops

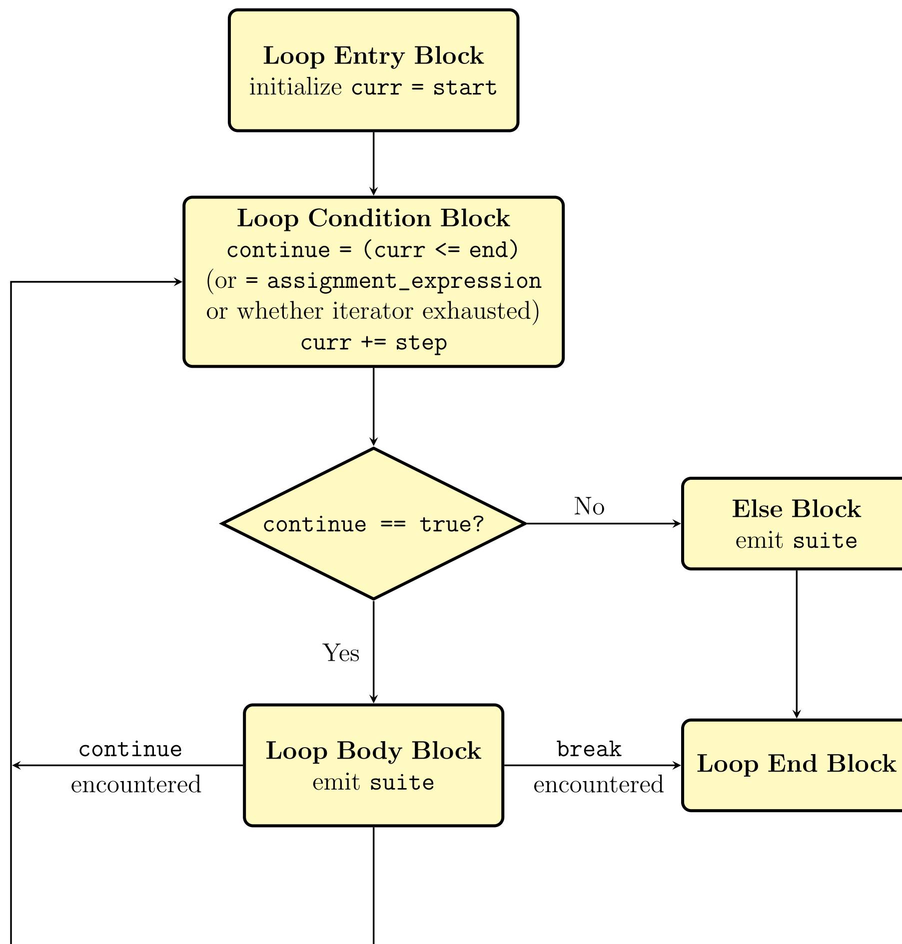

Figure 4·2: Code generation approach for a loop

Tuplex compiles UDFs into a series of basic blocks in LLVM IR. Figure 4·2 shows the process of code generation for a loop. We start at Loop Entry Block, where we initialize a counter `curr` to be equal to `start`. The `start` value is 0 for every loop except for looping over a range object, where `start` is then the start value of the range. Then, we enter Loop Condition Block to test if the loop should continue by

comparing `curr` with `end` for `for` loops (or by the boolean result from evaluating `assignment_expression` for `while` loops, or by check if the iterator is exhausted for iterating over an iterator). Before we exit this block, we also increment `curr` by `step`, which is a constant 1 for most loops or the range step for iterating over a range object. If the loops ends, we enter Else Block to emit `suite` under the else clause if there is one, and then enter Loop End Block to terminate the loop. If the loop should continue, we enter the Loop Body Block where the `suite` of the loop body gets emitted, and re-enter Loop Condition Block when every statements in the suite has been generated. If we encounter a `continue` or `break` statement in the middle, we would need to immediately jump either back to the Loop Condition Block, or directly to Loop End Block.

4.3 Iterators

Unlike primitive types, an iterator has internal states of which we need to keep track. We will use `list_iterator` which is returned by the `iter()` call on a list as an example to show how iterators are implemented. Structs for other iterators generated from `iter()` are similar, except that they will contain pointers to different iterable object types.

```
1 ListIteratorContext {
2   BlockAddress* lastBlock; // initialized to updateIndexBB
3   i32 index; // initialized to -1
4   ListStruct* listObject; // pointer to the list
5 }
```

Listing 8: LLVM struct for a `list_iterator`

In Tuplex, we use an LLVM struct similar to the pseudocode shown in listing 8 to represent a `list_iterator`. To allow `next()` to retrieve the next item from

a `list_iterator`, we use a helper function `updateListIteratorIndex()` to first determine if the `list_iterator` is exhausted. The logic of the function is shown in figure 4·3, where `lastBlock` is the field in the iterator struct with an initial value equal to Update Index Block's block address. Conceptually, the function increments an unexhausted iterator's `index`, and returns a bool value indicating whether the iterator is exhausted. If so, it also sets `lastBlock` to be the address of End Block, so that any future `next()` calls on this iterator will directly return `true`.[1]

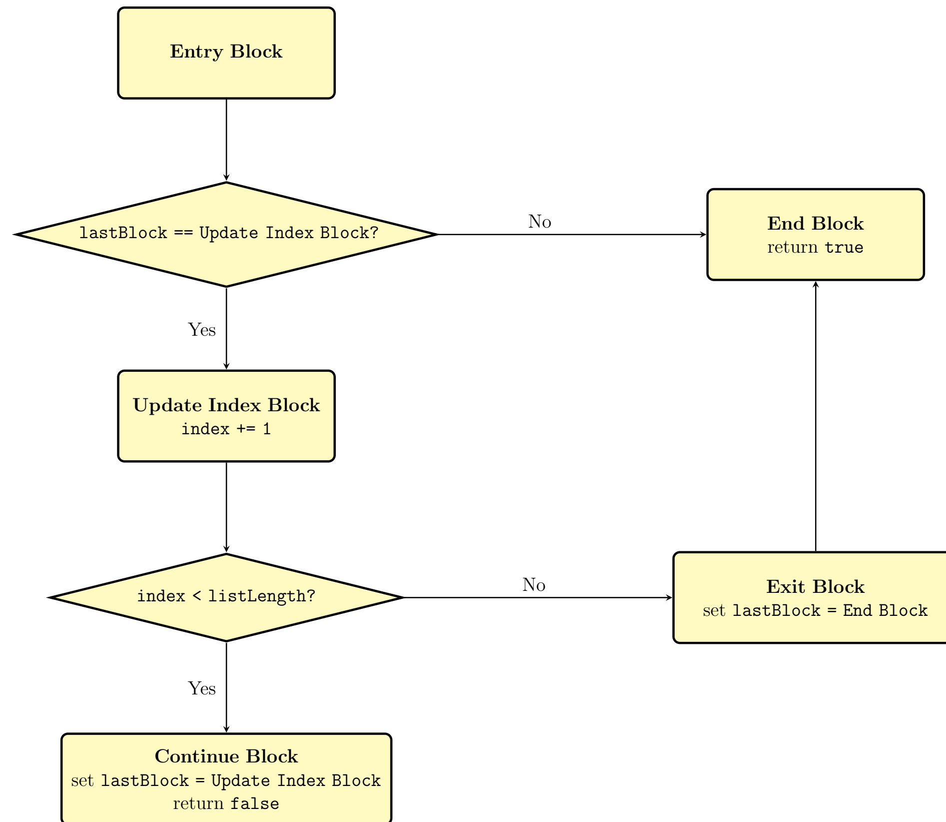

Figure 4·3: `updateListIteratorIndex`: Update a `list_iterator`'s `index` and check if it is exhausted.

Then, we retrieve next item from `*listObject` if the iterator is not yet exhausted. If it has run out of items, we generate code to raise an exception `stopIteration` or return the default value as shown in figure 4·4.

[1] This implementation is inspired by a strategy described by Rodler [18].

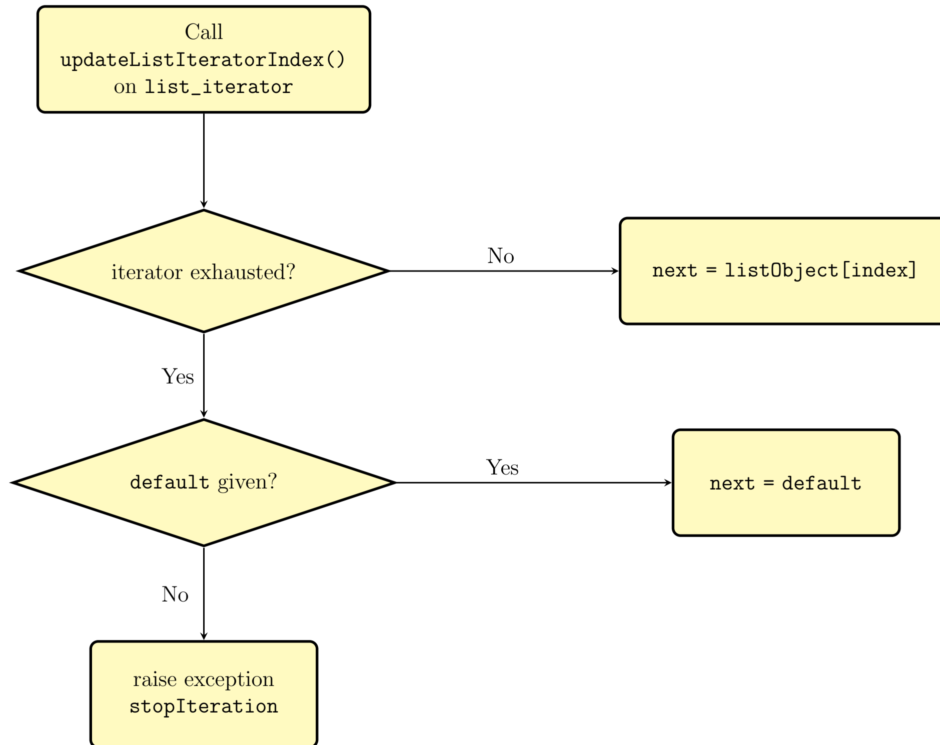

Figure 4·4: Calling `next()` on a `list_iterator`.

```
EnumerateIteratorStruct {
    i64 count   // initialized to -1
    ListIteratorContext*      argIterator;
}
ZipIteraotrStruct {
    ListIteratorContext*      argIterator1;
    StringIteratorContext*    argIterator2;
    EnumerateIteratorContext* argIterator3;
    ZipIteratorContext*       argIterator4;
}
```

Listing 9: LLVM struct for an `enumerate` iterator and a `zip` iterator.

Iterators returned by `enumerate()` or `zip()` are based on the iterators we covered above, but have slightly different LLVM structs and implementation. The pseudocode in listing 9 shows the LLVM structs for an `enumerate` iterator returned from

`enumerate([1, 2, 3, 4], -1)` and a `zip` iterator returned from a function call `zip([1, 2], "hello", enumerate("tuplex"), zip([1, 2], (3, 4)))`.

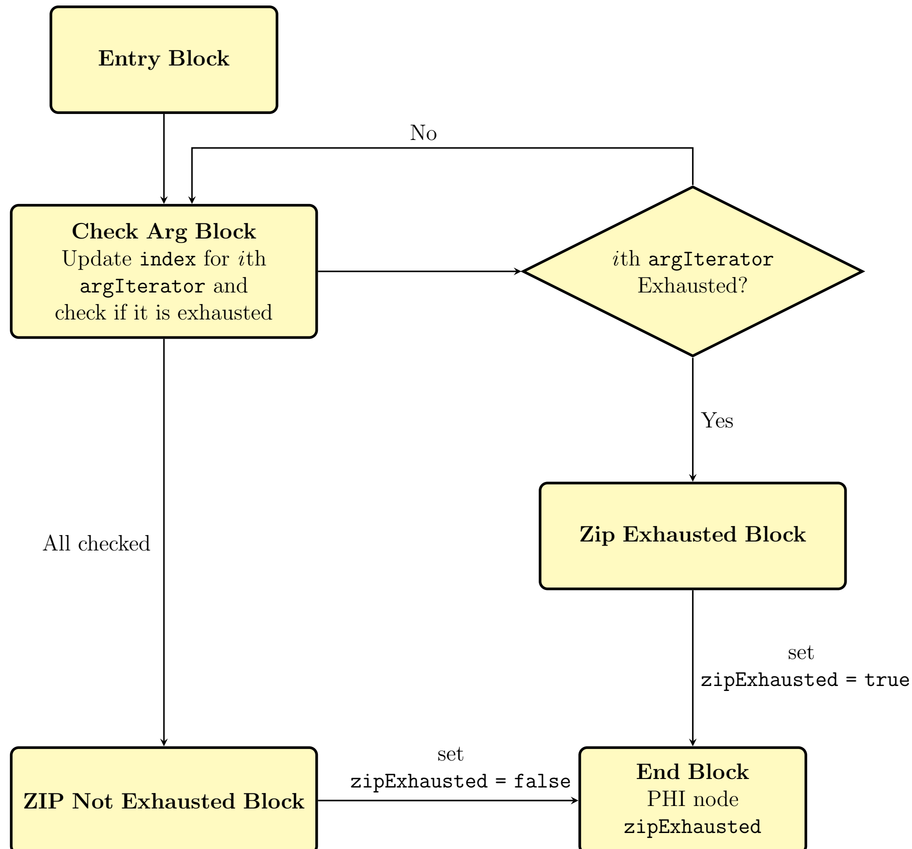

Figure 4·5: `updateZipIteratorIndex`: Check if a `zip` iterator is exhausted. The value of the PHI node `zipExhausted` depends on the incoming block before End Block.

The struct for a `enumerate` iterator has a pointer to another iterator `argIterator`, and a counter whose value will be returned along with the next item from `argIterator`. The struct for a `zip` iterator simply contains pointers to other iterator structs. Note that an `enumerate` or `zip` iterator can contain pointers to other `enumerate` or `zip` iterators.

A `zip` iterator uses a more complex `updateZipIteratorIndex()` function to determine if it is exhausted when `next()` gets called on it. Figure 4·5 shows the control

flow of the LLVM function. If `zipExhausted` is `false`, `next()` will then retrieve an item from each `argIterator` and return them as a tuple.

4.4 Loop Unrolling

To compile a UDF in Tuplex, static typing is attempted first by adding type annotations along the AST tree. Static typing may fail if different branches in an if statement have different types for the same variable or if there is a potential type stability issue in a loop.

Python Codes	**LLVM IR**	**Variable Slot Name PythonType Ptr**
`x = 10`	`%1 = alloca i64` `store i64 10, i64* %1`	x int %1 ⇩
`x = "hello!"`	`@s = private unnamed_addr constant [7 x i8] c"hello !\00"`	x ~~int~~ ~~%1~~ str @s

Figure 4·6: How Tuplex keeps track of a Python variable's type during code generation

Since Tuplex internally generates LLVM IR codes from Python source code, every time a variable's type changes, new memory needs to be allocated for the variable to reflect the new type. If the control flow is sequential, this is not a problem. Figure 4·6 shows an example of how Tuplex handles Python's dynamic typing. For every new variable encountered, an instruction to generate LLVM IR code that allocates new memory with the variable's corresponding LLVM type will be emitted. In addition, an entry with the variable's name, type, and memory location (Ptr) is added to Variable

Slot [2]. When the type of `x` changes from integer to string, we allocate new memory with its new corresponding LLVM type and overwrite its Variable Slot entry with the new type and memory location. However, in an LLVM IR loop, the same set of instructions are executed in each iteration. This can result in some instructions still referring to the old Ptr in future iterations after a variable's type gets updated.

4.4.1 Full Loop Unrolling

To execute a `for` loop over a tuple in listing 4, instead of simply updating the value at the initial memory allocated for `x`, we would need to allocate different memory for each type of `x` prior to the first iteration and specify in LLVM IR which pointer to use for `x` in each iteration. This leads to our solution of loop unrolling. Instead of code generating this logic directly, we unroll such a loop at the AST level. Listing 10 shows conceptually equivalent pseudocode to listing 4 after the loop unrolling.

```
1  def unrolled_loop_tuple(t):
2      # for i in ([1, 2], "hello", [-1, -2, -3, -4], "a, b"):
3      #     t += len(i)
4      # This loop can be unrolled to
5      i = [1, 2]
6      t += len(i)
7      i = "hello"
8      t += len(i)
9      i = [-1, -2, -3, -4]
10     t += len(i)
11     i = "a, b"
12     t += len(i)
13     return t
```

Listing 10: Unrolling a loop over a heterogeneous tuple

[2] where Tuplex keep tracks of the last type associated with a variable.

4.4.2 Partial Loop Unrolling

```
def f(t):
    x = 1  # value at ptr1 = 1
    for i in range(t):
        x = x + i  # value at ptr1 = value at ptr1 + i
        x = x * 0.4  # value at ptr 2 = DoubleTy(value at ptr1) * 0.4
    return x  # return value at ptr 2
```

Listing 11: Ignoring the type stability issue generates code that returns incorrect result.

A hard-to-notice case of type stability issue occurs when type change happens inside the loop body. When the piece of codes in listing 11 gets generated to LLVM IR, at line 2, a memory region pointed to by `AllocaInst *ptr1` with type Int64Ty is allocated and associated to `x`. At line 5, a new memory region `AllocaInst *ptr2` with type DoubleTy is allocated for `x` and re-associated to `x`. This causes a problem, as values at two memory addresses are being updated despite belonging to the same variable: In every iteration of the loop, line 4 corresponds to the update for the old `x` at `ptr1`, but line 5 corresponds to the update for the new `x` at `ptr2`. In the end, since `x` was last associated with `ptr2`, value at `ptr2` will be returned although it is not the desired result.

To generate correct LLVM IR codes for this loop in Tuplex, we partially unroll the loop, separating the first iteration with the rest during code generation. The effect would be equivalent to the following pseudocode in listing 12 (suppose `t` is a positive integer for all inputs).

Now, `x` gets upcast to float before entering the loop, and all updates to `x` during the loop will only be written to the new memory region with type DoubleTy. In the end, a double value at `ptr2` will be returned as expected.

```
def f(t):
    x = 1  # value at ptr1 = 1
    # 1st iteration
    x = x + 0  # value at ptr1 = value at ptr1 + 0
    x = x * 0.4  # value at ptr2 = DoubleTy(value at ptr1) * 0.4
    # rest of the loop
    for i in range(1, t):
        x = x + i  # value at ptr2 = value at ptr2 + i
        x = x * 0.4  # value at ptr2 = value at ptr2 * 0.4
    return x  # return value at ptr 2
```

Listing 12: Partially unroll a loop with variable type change to resolve type stability issue

4.5 Type Tracing

```
def g(t):
    x = 1
    for i in range(1, t):
        if i % 5 == 0:
            x = 1
        else:
            x = "multiple of 5"
    return x
```

Listing 13: Example where partially unrolling does not work since the type of `x` is not stable after the first iteration.

For such a loop where variables' types are stable after the first iteration, we can compile it by unrolling the first iteration. However, for the example in listing 6, whether there is a type change in the loop depends on the input. And we should unroll the loop only if we know that the majority of the inputs will trigger the type change. In addition, unrolling will not always help resolve the type stability issue;

specifically, unrolling is insufficient if variables' types are constantly changing during the loop. An example above in listing 13 shows a loop that does not have type stability (suppose `t` is a positive large integer for all inputs) due to the type of `x` getting changed in every 5th iteration.

For the loop in listing 13, separating the first iteration will not work since the type of `x` is not stable after the first iteration. In fact, it is constantly changing between integer and string. By statically analyzing the AST, we can tell if any variable's type has changed during loop, but hard to know if the type change results in type stability from the second iteration. In such a case, traced typing is used to predict type stability (Figure 4·1). We sample rows from the parent dataset as inputs and use the Python/C API to execute the UDF within the Python interpreter to gather necessary typing information. Specifically, for each sampled input, we record if each loop in the UDF is (i) indeed has a type change and (ii) type is stable after the first iteration. If most samples do not trigger a type change for a loop during tracing, we will compile the loop without unrolling, but raise a `NormalCaseViolation` exception if some exceptional rows trigger type changes and send them to fallback path. If most samples result in unstable types after the first iteration in a loop, we will speculate that the loop would not compile for majority of the inputs, and execute the UDF through the fallback path. Otherwise, we will generate a normal path using partially unrolling if most samples prove type stability after the second iteration. For the minority rows that violate the normal case, `NormalCaseViolation` will be raised, and they will be executed through the Python interpreter.

Chapter 5

Evaluation

5.1 Setup

In this chapter, we show some results from evaluating Tuplex, CPython, Numba and Pypy on two Python UDFs that contain loops in terms of running time. We used Python version 3.9 anywhere Python was involved. The Tuplex version used was 0.3.3rc0, and the Numba version used was 0.55.1. All experiments were run on macOS Monterey 12.0.1 with 16 GB memory and a 2.8 GHz Quad-Core Intel i7 process.

5.2 Experiment I: Numeric-heavy workload

Listing 14 is a UDF adapted from a blog post on Towards Data Science [19] that counts prime numbers up to an upper bound. This workload is numeric-heavy, and serves as a baseline experiment.

We used CPython, Tuplex, Numba, Pypy, as well as C++ to complete the same task of computing `count_prime` on 200 inputs (integers between [10, 100] with a step size 10, repeated 20 times). For CPython and Pypy, we looped over the input list and ran the function for each integer in the list. For the Numba version, we used the same Python code except that the decorator `@jit(nopython=True)` was applied to the `count_prime` function. For Tuplex, inputs were represented as 200 rows, and

the `map` operation was used to apply `count_prime` to each row. We also hand-wrote C++ to complete the same task and recorded the C++ running time. The C++ version was compiled using Clang with `-O3 -march=native` option.

```
def count_primes(max_num):
    # This function counts prime numbers below the input value.
    # Input values are in thousands, i.e., 40, is 40,000.

    count = 0
    for num in range(max_num * 1000 + 1):
        if num > 1:
            for i in range(2, num):
                if num % i == 0:
                    break
            else:
                count += 1
    return count
```

Listing 14: A UDF that counts prime numbers within an upper bound.

In the single-threaded test (with running time shown in figure 5·1 and figure 5·2), Tuplex is 26.70x faster than the CPython interpreter and is only 13% slower than C++. Tuplex also outperforms Numba and Pypy by 1.93x and 2.23x, respectively.

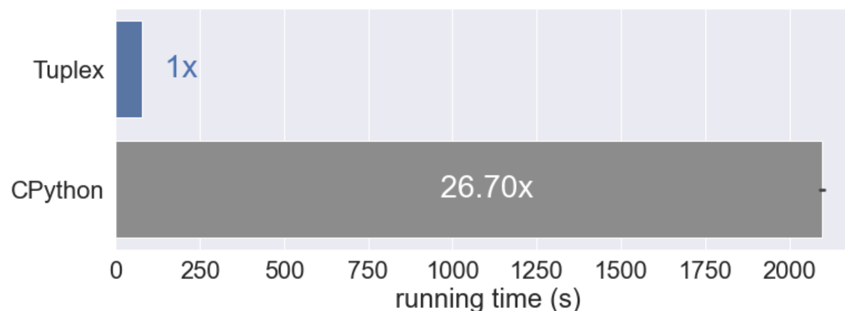

Figure 5·1: Tuplex vs. CPython on `count_primes`, single-threaded.

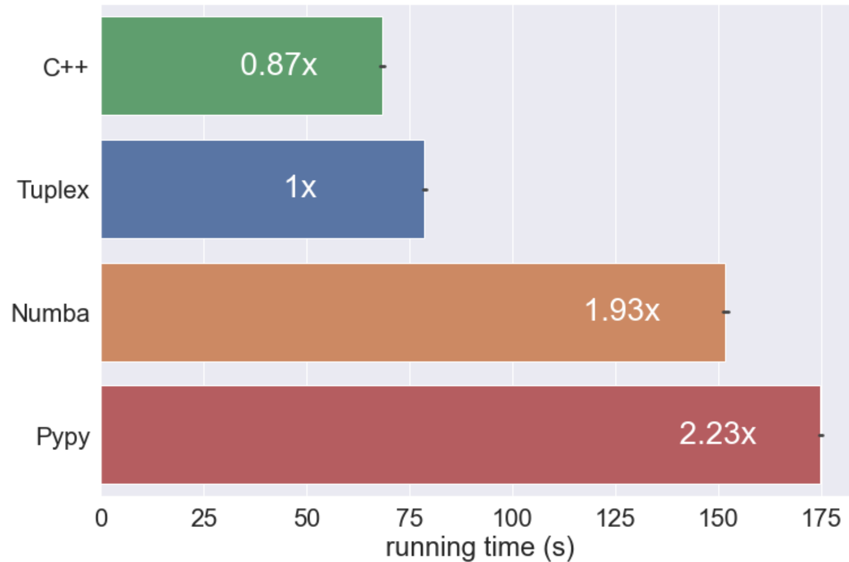

Figure 5·2: Tuplex vs. C++, Numba and Pypy on `count_primes`, single-threaded.

For comparing the performance of completing the same task in a parallel setting, the CPython and Pypy version used the `multiprocessing` package and created a `Pool` of 4 worker processes to split the task. The C++ version used 4 threads. The Tuplex version used 4 executors [1]. For Numba, `set_num_threads(4)` was called, `parallel=True` was added to the function decorator, and `prange()` was used instead of `range()` in the UDF to enable parallel loops.

The benchmark with parallelism produced similar results as for the single-threaded benchmark (figure 5·3 and figure 5·4). Tuplex is more than twice as fast as Numba and Pypy.

[1]To have parallelism among executors, we need more than 1 partition, so `partitionSize` was set to only fit 10 integers to produce 20 partitions out of the 200 rows.

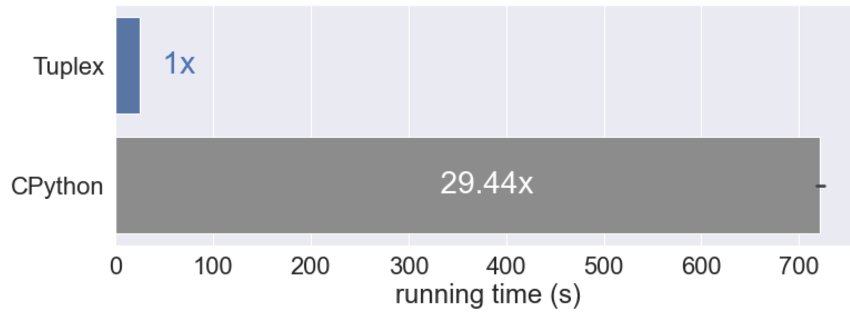

Figure 5·3: Tuplex vs. CPython on `count_primes`, 4-way parallelism.

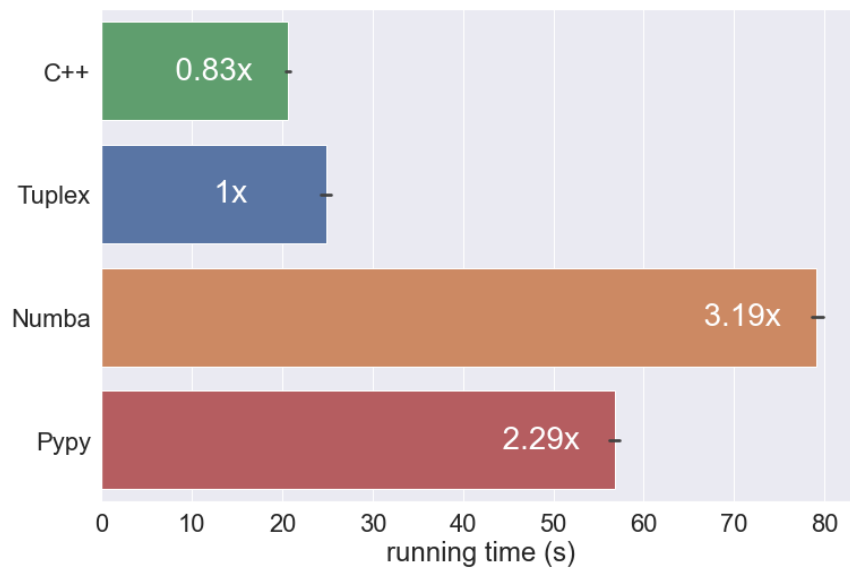

Figure 5·4: Tuplex vs. C++, Numba and Pypy on `count_primes`, 4-way parallelism.

5.3 Experiment II: Experiment where Numba is better

```
def loop_with_iterator(x):
    y = 0
    zip_iter = zip(range(0, x*x , 7), range(0, -x*x, -3))
    for i, j in zip_iter:
        if (i - j) % 3 == 0:
            y += 1
        if (i - j) % 5 == 0:
            y -= 1
        if (i - j) % 7 == 0:
            y += int(math.sqrt(i + j))
        if (i - j) % 11 == 0:
            next(zip_iter)
    return y
```

Listing 15: A UDF that involves iterators.

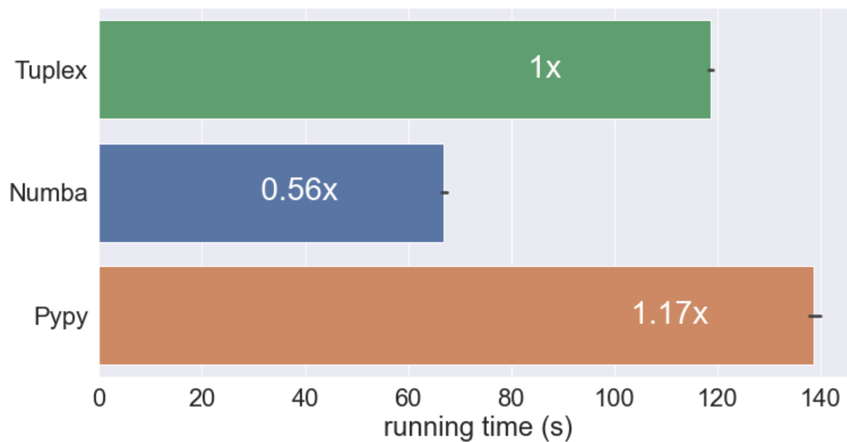

Figure 5·5: Tuplex vs. Numba and Pypy on `loop_with_iterator`, single-threaded.

Using similar setup [2] and with 100 inputs ([10000, 20000, 30000, 40000, 50000] repeated 20 times), we tested UDF in listing 15 on CPython, Tuplex, Numba, and

[2]Numba's `prange()` does not support step size larger than 1 and thus cannot run `loop_with_iterator` in parallel. For the multi-threaded version, Numba was actually run under Python's multi-processing module with 4 workers.

Pypy. Tuplex achieves more than 40x speed up compared to CPython on both single-threaded version and multi-threaded version. The comparisons among Tuplex, Numba, and Pypy are shown in figure 5·5 and figure 5·6. In both versions, Tuplex is around 1.2x better than Pypy, but is 50% slower than Numba. A possible explanation is that unlike in `count_primes` where there is not much optimization to be done and a loop would need to go through each iteration to correctly compute the count, `loop_with_iterator` allows for more optimization. Tuplex's loop compilation mostly relies on LLVM for further optimization, but Numba may have more specialized optimization towards certain loops.

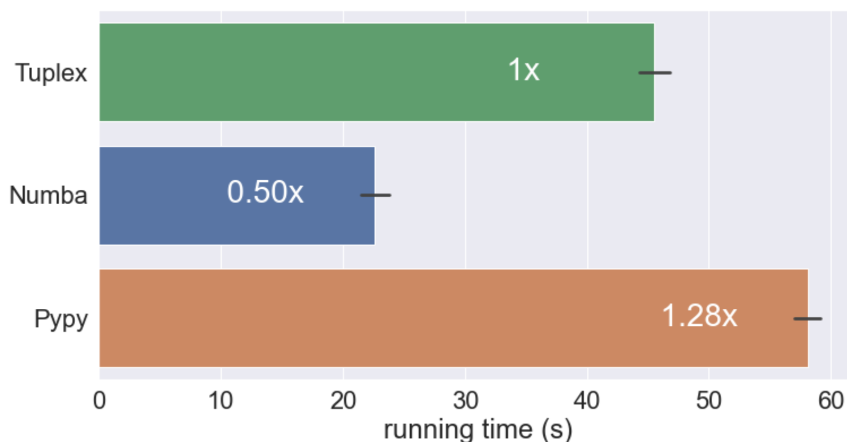

Figure 5·6: Tuplex vs. Numba and Pypy on `loop_with_iterator`, 4-way parallelism.

5.4 Numba Limitation

For a loop similar to listing 16, Tuplex uses type tracing to decide the return type for `x`. If most inputs are non-positive, the common case should be returning an integer 0 since the loop will likely not run. However, in Numba's generated LLVM IR code, `x` will always be upcast to a float before entering the loop. Even when the function input is 0 or a negative number, it will return a float 0.0 instead of an integer 0, which

sometimes may not be what a user would expect.

```
def loop_with_type_change(n):
    x = 0
    for i in range(n):
        x += 1
        x += 1.0
    return x
```

Listing 16: A UDF that contains a loop with type change.

Chapter 6

Conclusions and Future Work

In this thesis, we talked about our contribution of adding loop and iterator support to the Tuplex project. We presented how we resolved the type stability issue with loop unrolling and type tracing. Our implementation enables Tuplex to achieve over around 26x speedup for UDFs containing loops, and outperforms some popular Python compilers or implementations by more than 2x in executing certain loops. Future work includes supporting a more comprehensive loop syntax (e.g., using `_` as `target_list` and using dictionaries as `expression_list` in `for` loops), exploring more compiler optimization methods to speed up loop execution, and exploring methods that enable compilation for a subset of Python loops with more complex type changes.

References

[1] Leonhard Spiegelberg, Rahul Yesantharao, Malte Schwarzkopf, and Tim Kraska. Tuplex: Data Science in Python at Native Code Speed. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, page 1718–1731, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3448016.3457244.

[2] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. PEP 484 – Type Hints. https://peps.python.org/pep-0484/. Accessed May 1, 2022.

[3] Albert van Breemen. NYC Taxi Fare - Data Exploration. https://www.kaggle.com/code/breemen/nyc-taxi-fare-data-exploration, July 2018. Accessed May 1, 2022.

[4] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-Based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery. https://doi.org/10.1145/2833157.2833162.

[5] Numba. Supported Python features. https://numba.readthedocs.io/en/stable/reference/pysupported.html#supported-python-features. Accessed May 1, 2022.

[6] Pypy. Goals and Architecture Overview. https://doc.pypy.org/en/latest/architecture.html. Accessed May 1, 2022.

[7] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, page 18–25, New York, NY, USA, 2009. Association for Computing Machinery. https://doi.org/10.1145/1565824.1565827.

[8] LLVM. The LLVM Compiler Infrastructure. https://llvm.org. Accessed May 1, 2022.

[9] Michael Rodler. Local Variables. https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/en/latest/basic-constructs/local-variables.html. Accessed May 1, 2022.

[10] Python 3.9.12 Documentation. List Comprehensions. https://docs.python.org/3.9/tutorial/datastructures.html#list-comprehensions. Accessed May 1, 2022.

[11] Guido van Rossum. From list comprehensions to generator expressions. http://python-history.blogspot.com/2010/06/from-list-comprehensions-to-generator.html, June 2010. Accessed May 1, 2022.

[12] Python 3.9.12 Documentation. Assignment expressions. https://docs.python.org/3.9/reference/expressions.html#assignment-expressions. Accessed May 1, 2022.

[13] Python 3.9.12 Documentation. The for statement. https://docs.python.org/3.9/reference/compound_stmts.html#the-for-statement. Accessed May 1, 2022.

[14] Python 3.9.12 Documentation. The while statement. https://docs.python.org/3.9/reference/compound_stmts.html#the-while-statement. Accessed May 1, 2022.

[15] Python 3.9.12 Documentation. Iterator Types. https://docs.python.org/3.9/library/stdtypes.html#iterator-types. Accessed May 1, 2022.

[16] Python 3.9.12 Documentation. next. https://docs.python.org/3.9/library/functions.html#next. Accessed May 1, 2022.

[17] Eric W. Weisstein. Collatz Problem. From MathWorld–A Wolfram Web Resource. https://en.wikipedia.org/wiki/Collatz_conjecture. Accessed May 1, 2022.

[18] Michael Rodler. Generators. https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/en/latest/advanced-constructs/generators.html. Accessed May 1, 2022.

[19] Thuwarakesh Murallie. How to Speed up Python Data Pipelines up to 91X? https://towardsdatascience.com/how-to-speed-up-python-data-pipelines-up-to-91x-80d7accfe7ec, July 2021. Accessed May 1, 2022.