Compilation Techniques for Distributed Analytics

Anna Herlihy
Undergraduate Thesis
Computer Science, Sc. B.
Brown University 2014
Advisor: Tim Kraska
Reader: Ugur Cetintemel

ABSTRACT

Tupleware is a new analytical framework built at Brown University that allows users who wish to run algorithms on large datasets to have their functions compiled into distributed programs and automatically deployed. Currently, the user is limited to writing their algorithms in C++, which is then compiled into LLVM-IR. The LLVM Compiler Infrastructure Project provides a convenient, transportable intermediate representation (LLVM-IR) which can be compiled and linked into multiple types of machine-dependent assembly code. For this thesis, a comprehensive Python frontend was written to allow the user to access Tupleware and write their own algorithms in Python. The goal of the thesis is to explore the possibility of seamlessly integrating high-level languages with Tupleware using LLVM and to demonstrate various compilation techniques involved in our Python-to-LLVM compiler, PyLLVM.

Table of Contents

I Introduction

Motivation

The goal of this project is to make the user's experience with the Tupleware framework as simple and straightforward as possible through providing an interface with the C++ backend. With various technological advances, we are now able to collect data on a much larger scale. Machine learning is becoming an extremely important and widely used tool for the analysis of large datasets. This thesis describes the design and implementation of Python to LLVM compiler specialized for machine learning computations in Tupleware.

The primary usage of Tupleware is running machine learning algorithms on large datasets. The nature of core ML algorithms are that they are often simple mathematical expressions, which can easily be optimized. They often do not require complicated functionality outside regular mathematical functions. At this point, Tupleware is only capable of processing user-defined algorithms in C++.

Python

```
def naive_bayes(data=listfx[1], counts=listfx[1], dims=int, vals=int, labels=int):
    label = data[dims]
    counts[label] =+ 1
    offset = labels + label * dims * vals
    while(c in range x[1]):
        counts[offset+c*vals+data[c]]=+1
```

[1]In the full implementation these would be constants defined by the user.

C++

```
void naive_bayes(char *data, int *counts, int dims, int vals, int labels) {
    char label = data[dims];
    ++counts[label];
    int offset = labels + label * dims * vals;
    for (int j = 0; j < dims; j++)
        ++counts[offset + j * vals + data[j]];
}
```

Figure 1: Naive Bayes Algorithm in Python and C++

Currently, the two most frequently used languages for data science are MATLAB and Python. Requiring the user to write their code in C++ can be burdensome. Because C++ requires explicit memory management, the user is charged with keeping track of their own memory usage which is not the case for MATLAB and Python. Since ML algorithms rarely need to do any fancy memory management, it is

easier for the user to not have to deal with pointers and references. Additionally, C++ does not provide high level native data types (strings, tuples, lists, dictionaries, etc.) that are available in MATLAB and Python. A goal of this thesis is to provide the user with a Python interface to Tupleware, which includes the option of writing the algorithms that will be deployed in Python.

For example, consider the naive Bayes algorithm. Figure 1 is an implementation of naive Bayes in Python and C++. While the two versions look similar, the semantics of the programs are such that there is no functionality unique to C++ that is required: for example pointers, multithreading, etc. Since both functions will compile down to similar LLVM, runtime is also not an issue. Python offers a plethora of advantages over C++ without imposing more limitations than C++.

Why Python

Python is preferable over more scientific languages like MATLAB because of its simple syntax and readability. There are many quirks of MATLAB syntax that make it harder to convert code from other languages to MATLAB, such as one-based indexing and parenthesis instead of brackets for indexing into arrays. MATLAB also uses 'end' statements to implement closures, which Python avoids by enforcing indentation in blocks. Importing other files is also much more complicated in MATLAB than in Python, since MATLAB checks only in the MATLAB path and the file itself. Additionally Python is free and open source while a MATLAB licence is relatively expensive. Generally, Python code is almost always shorter than the equivalent algorithms in MATLAB. Simpler syntax contributes to compact code, faster debugging, and easier understanding of other's code. Lastly, from an implementation point of view, Python provides a compiler package that is well documented and very easy to use that can produce an AST and the ability to traverse it with a visitor class.

Steps

The hypothesis of this thesis is that it is possible to tightly integrate higher-ordered languages, such as Python, into a general purpose distributed analytical framework, like Tupleware, without sacrificing the usability of the language. There were three distinct steps in the process of integrating Python with Tupleware. First, the user needed to access the Tupleware backend from Python. A Python wrapper was written to access the Tupleware C++ frontend functions using Boost.Python. These functions require user-defined functions to be input as LLVM, so the second step was to convert the user's Python code to LLVM. There were a few options for the generation of LLVM which are discussed in detail in the **Related Work** section of this paper, but the choice was made to write a new

Python-to-LLVM compiler based on an existing abandoned google project "py2llvm". Continuum Analytics provides Numba, a just-in-time Python-to-LLVM compiler, which we chose not to use because we wanted a small, simple static compiler which we could mold to our specific needs and modify at will. The third step was to provide a backup mechanism in case our compiler was unable to process the user's code. If the user input code that we were unable to compile but was still valid Python code, we convert the Python to C++ using the Python-C API. Once the Python is converted to C, it can then be passed to the existing Tupleware frontend to be run. Once the user-defined code is converted to LLVM, the user is able to call the Tupleware operators in order to have their code automatically deployed on the distributed framework. The remainder of this thesis is structured as (1) an explanation of Tupleware, (2) a description of the process involved with integrating Python and Tupleware, (3) an analysis of the results, (4) related work, and (5) conclusion and future work.

II Tupleware

Tupleware was developed to investigate the impact of small enterprise clusters with fast interconnects, such as Infiniband FDR or Ethernet 100G. With these technologies, the bandwidth available for transferring data across this network is in the same ballpark as the bandwidth of the memory bus. Although latency is still slower, it makes memory locality insignificant for data-intensive operations and the only "real" bottlenecks left are the CPU cache and CPU efficiency. The core idea is to leverage modern compiler techniques as much as possible to best take advantage of the CPU cache and modern CPU features by compiling all queries and in-line user-defined functions (e.g., ML algorithms) into distributed programs and automatically deploying them.

Tupleware Architecture

Figure 2 describes the architecture of Tupleware. Ultimately, the goal of Tupleware is threefold: (1) provide a language-independent front end, (2) tightly integrate UDFs with control flow statements, and (3) automatically synthesize optimized distributed programs for deployment in a cluster. Users are able to define their own workflows by combining operations (e.g. joins, maps) from the Tupleware library and passing in the user-defined functions. Once the user has submitted a job, there are four distinct steps (see the four middle boxes of Figure 2). First, the entire workflow as well as each individual UDF is analyzed with the **Function Analyzer**. Second, **The Optimizer** then generates a logical execution plan

(LEP) that is able to take advantage of modern hardware features available on each node in the cluster. Specifically, we introduce optimizations for selections, consecutive map operations, and aggregations (i.e., combines and reduces in the MapReduce paradigm). **The Scheduler** then determines how to best deploy the job given the available resources, and finally the **Code Generator** transforms, optimizes and compiles the LEP into executables that are executed on nodes in a cluster. See Figure 2.

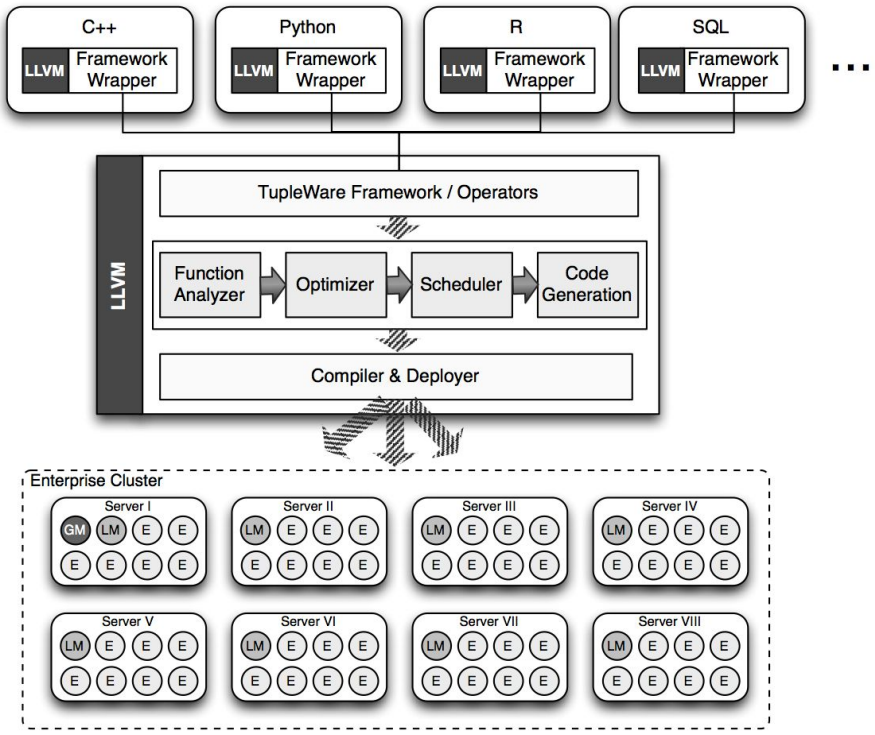

Figure 2: Tupleware Architecture

Tupleware Operators

There are five types of operators available in the Tupleware frontend. The first three types of operators (1) map-like, (2) reduce-like, and (3) join-like, allow Tupleware the flexibility of MapReduce while retaining the optimization potential of SQL by extending the MapReduce paradigm with a join-style operation. The map-like operators include `map, flatMap, filter;` the reduce-like operators include `reduce, combine, apply`; the join-like operators include `join` and `cartesian`. The next type of operator, (4) control-like, which includes `loop`, is intended to allow support for iterative algorithms. Lastly, the (5) command-like type of operator allows the user some imperative control over the data if, for example, they

need to be able to persist data to a non-volatile storage device. The command-like type of operators include `eval`, `save`, and `peek`.

The operators are defined on a data structure called `TupleSet`. As in any functional language, all transformations must be side effect free; that is, they cannot have any observable interaction with the outside world. The operators are described in Figure 3.

Class	*Map*	*Reduce*	*Join*	*Control*	*Command*
Operator	`map, flatMap, filter`	`reduce, combine, apply`	`join, cartesian`	`loop`	`eval, save, peek`

Figure 3: Tupleware Operators

An example execution using these operators can be found in Figure 4.

How Frontend Languages are Integrated

In order to integrate a new language into the Tupleware framework, the user needs to be able to, in their language of choice, (1) access the tupleware operators and (2) compile their UDFs into LLVM. The first step is often simple, as for many languages there are wrappers written to allow calling C++ code directly. The second component is a little more complicated, as even though many LLVM compilers exist, making them callable from the wrapped Tupleware library is not always trivial. For this project, since we were writing our own compiler in the frontend language (Python), it was as simple as importing the compiler library (PyLLVM) into TupleWrapper (the Python wrapper for Tupleware).

III Integrating Python and Tupleware

Approach

The full Python frontend for Tupleware requires three distinct components. The first step is to allow Tupleware to be accessed from Python, which is handled in **TupleWrapper**. The second step is to convert UDFs into LLVM from Python, which is handled by **PyLLVM**. The last **backup** step is used if the user provided valid Python that PyLLVM was unable to compile, which converts the UDF into C using the Python-C API and passes the C code directly to the existing Tupleware frontend. A simple example Python execution can be found in Figure 4.

Example Python Tupleware Execution

```
import TupleWare
def run(udf, data):
    TS = TupleWare.load(data)
    TS.map(udf)
    TS.execute()
```

Figure 4: Tupleware Python Execution

1. TupleWrapper

The topmost layer is the Python wrapper of the C++ operators in `TupleSet`. The Boost library provides a simple interface for creating a Python library file from a C++ object. The C++ file TupleWrapper.h, which contains code that wraps the TupleSet.h functions using Boost Python was added to the Tupleware source. The Tupleware build was modified so that every time Tupleware was recompiled, a Python object file TupleWrapper.so is generated. This file provides a Python library that calls the `TupleSet` functions. With access to this file, the `TupleWrapper` library can be imported in Python and any of the Tupleware frontend operators can be called, passing the required values as they would with any native Python library. The code for wrapping TupleSet.h can be found in src/frontend/TupleWrapper.h, and the resulting object file is found as build/bin/TupleWrapper.so after calling make.

Since TupleWrapper is automatically generated and calls the C++ frontend function directly, the functions that take in UDFs, for example `map`, expect the input as LLVM. In order to seamlessly integrate the Python compilation and the calling of the frontend functions from TupleWrapper, a `Tupleware` library was created in Python. This library provides direct access to all the `TupleWrapper` functions, except the ones that take in LLVM IR as arguments. These functions are modified to expect UDFs in Python, which are run through the Python-to-LLVM compiler and the result is sent to the original TupleWrapper function. For example, the `map` function of the TupleWare library can be found in Figure 5.

The `map` Function of TupleWare

```
import PyLLVM
import TupleWrapper
def map(udf):
    try:
        # try to get LLVM from PyLLVM
        llvm = PyLLVM.compiler(udf)
    except PyLLVM.PyllvmError as e:
        # PyLLVM was unable to compile the UDF, now try backup method
        TupleWrapper.backup_map(udf)
    except Exception:
        # the exception was semantic, so the UDF contains invalid python
        raise Exception("Bad Python in UDF")
    else:
        # valid LLVM IR was generated, and is now being passed to the backend
        TupleWrapper.map(llvm)
```

Figure 5: TupleWare `map`

2. PyLLVM

The primary contribution of this thesis is the PyLLVM compiler, which converts functions written in a subset of Python to LLVM. The goal of this part of the project was to write a simple, easy to extend, one-pass static compiler that takes in a subset of Python most likely to be used by Tupleware user-defined functions. The closest existing project was Google's py2llvm project, which was written and subsequently abandoned sometime around 2010. Unfortunately, since the project was never finished, none of the features implemented were fully functional. Additionally, there was no usable documentation available for the existing code (the only thing available outside of code comments was a slide show in Japanese). This project fixes and extends the py2llvm project code to be a fully functional compiler for a subset of Python.

Subset of Python

The subset of Python covered by this compiler was chosen based on the anticipated common requirements for Tupleware users. Since the users will be most often writing machine learning algorithms, which are often small and require simple mathematics, the subset does not include any of the more advanced Python functionality: objects, decorators, generators, or iterators beyond iterating over lists. The lack of coverage for objects is also due to the nature of LLVM, as implementing objects in LLVM is a complex exercise that would greatly complicate the compiler code. Instead, primarily statically type-inferable code is handled.

Overview of Design

Python provides a compiler package for analyzing Python source code and generating Python bytecode. The package provides a Python parser, `compiler.parse(buf)`, that will generate an in-memory representation of a Python abstract syntax tree from the text found in `buf`. It also provides a `compiler.walk(ast, visitor)` function that will do a pre-order walk over the AST calling the corresponding node function in `visitor`.

Once the UDF is parsed, the `CodeGenLLVM` visitor class is applied to the AST and LLVM code is generated. The `SymbolTable` object is defined externally to keep track of the variables and scope, as well as the `TypeInference` object which is used to infer types for Python expressions. The `CodeGenLLVM` visitor is constructed in the classic "visitor" pattern, and passes through the AST once (with one exception for inferring return types, which is detailed in the *Functionality and Implementation* section below).

Code generation is handled through the LLVM IR-Builder. The LLVM compiler framework, which is written in C++, is accessed through the `llvmpy` package. Llvmpy is a Python wrapper around the LLVM code generation library by Continuum Analytics that provides access to the LLVM IR-Builder module. A string representation of the generated LLVM IR can be extracted from the builder module and returned at the end of compilation.

Functionality and Implementation

Scoping and Variables

SymbolTable

The `SymbolTable` module keeps track of stored variables and functions and their scopes. The `Symbol` class represents a variable or a function with a name, type, and memory location. In the case of an array, since the size is variable the `Symbol` will also keep track of the dimensions of the array. The symbol table itself is a stack of tuples, which represent scopes. Each tuple has a string (the name of the scope, i.e. "global") and a map of string (the symbol name) to symbols. Since the symbol table is implemented as a stack, it makes it very easy to keep track of function variables since when entering and leaving a scope, the variables can be pushed and popped off the stack as a set. Since the symbol table contains maps of names to symbols, the lookup time for a variable is only limited by the number of scopes in the symbol table.

SSA

LLVM is a Static Single Assignment based representation, which for our purposes essentially means that variables can only be assigned to once. Instead of implementing the entire compiler in SSA form, which would be incredibly confusing, variables are all allocated on the stack and their addresses stored in the symbol table. When a variable name is encountered, it's value is accessed through loading from the address stored in the symbol table. This allows the compiler to essentially treat variables as if they are reassignable. However since LLVM is not dynamically typed, dynamic reassignment is not supported although it is definitely possible to add this functionality through manipulation of the symbol table and more careful tracking of scope. Due to the nature of the anticipated user-defined functions, the benefit of allowing dynamic reassignment did not seem worth the necessary complex implementation.

Types

TypeInference

Since LLVM is statically typed, it is necessary to be able to infer the types of expressions in advance. The `TypeInference` module is a mechanism for inferring Python types from nodes of the Python AST. Depending on the name of the node, the name of the corresponding inference function is extracted and called. The `inferType` functions recursively descend the tree until it reaches a leaf node, which will indicate the type of value for the Python expression. The majority of nodes are simple (i.e. constants). Names are looked up in the symbol table, and function calls are checked against both the symbol table for user-defined functions as well as the set of intrinsic functions. For intrinsic functions in particular, to avoid needing different functions for integer and floats (i.e. fabs and abs), intrinsic math functions return the type that they are passed in.

Vectors

PyLLVM provides a 4-element immutable floating point vector type, for which vector math functions add, subtract, multiply, divide, and compare are supported. These are implemented with specific

ML needs in mind, since vectors are fairly common for mathematical analysis. It is also quite easy to implement variable length vectors, although for that functionality lists may serve the user better since they are less limited. The LLVM framework provides vector types that can keep track of only constants (integers and floats) - you cannot have a vector of vectors or a vector of pointers to non-constant values. Unlike Arrays, vectors are not passed by pointer. Additionally, vector values can be accessed through regular indexing (brackets) as well as '`vec.x/y/z/w`' for their respective values. The vectors used in the compiler (since they are not native Python types) are defined in VecTypes.py.

Lists

PyLLVM provides static-length mutable list types. LLVM provides array types that can be populated with constants or pointers. Because pointers to lists also have to have explicit dimensions, dynamically resizable lists are not currently implemented. Once dynamic variable reassignment is implemented, dynamically resizable lists would be easy to add.

It is possible to construct lists explicitly (`x=[1,2,3]`) or by calling `range(start, end)`, which creates a list and populates it with increasing integers or floats from the `start` value to `end`. There is also the `zeros` function, which takes as the first argument the length of the list to be created and as the second argument the value the list will be populated with (zeros if no arguments are passed in). This is mostly a helper function for generating test data, and is not present in native Python. Lists are `alloca_array`'d onto the stack, and are passed to functions by pointer. The only issue with lists arises when returning a newly created array from a function. Because lists are passed by pointer, lists defined on the stack within a function will be out of scope when the function returns. The temporary solution to this issue is to malloc arrays that being returned. These are the only values that are ever stored on the heap.

It is possible to iterate over a list using a for loop. There are also two intrinsic list functions, `len(L)` which will return the length of a list and `range`, which can take either 1 or 2 arguments of either int or float type. If you pass in floats to range, the list will be populated with floats and same for integers. Indexing into lists is supported through brackets, which can also be used to assign to specific locations in the list. Currently, lists of pointers to other lists are not supported although it would very easy to add that functionality (it is partially implemented).

String

Since strings are essentially lists of characters, and characters are integers, strings are implemented as wrappers around lists of integers. The symbol table keeps track of if a given list variable contains integers or a characters. The primary difference occurs in the print function, where in the case of a string each character is converted back into a char to be printed.

Integers/Floats

Numerical values are implemented as either 32-bit integers or 32-bit floating point. Boolean values are generated as 1-bit integers, but are later converted to integers to be stored. Booleans are interchangeable with integers 1 and 0 (which is similar to Python - i.e. True+True=2).

Functions

User Defined Functions

Users are able to define their own functions and call them from anywhere within their UDF. A function signature is generated and arguments are added to the symbol table for that function's scope when the function definition is visited. The only time this compiler visits a subtree twice is to infer the return types of functions. When the visitor reaches a function definition node, it does one descent where it extracts the return type (since function return types cannot be predeclared in Python). It then discards the changes (through popping the symbol table scope and calling delete on the builder) and runs the pass again.

Arguments passed to functions must have default values, since types are not dynamic. An example: `func(i=int, f=float)`. The only difference between regular Python argument passing and the subset covered in this project is that lists must have their dimensions specified in the default value. In regular Python, to pass an argument with a default value of a list is enough to say `i=list` or `i=[]`. In this subset, the type and length of the list must be appended to the default value. For example, for an array of integers of length 8, the default argument would be `i=listi8`. In order for this to run with the official Python interpreter, the user must define the default argument at the top of the file: `listi8=[]`. This is a minor hindrance for the user, and in future versions of this compiler it should be possible to infer the types and dimensions of lists.

Intrinsic Math Functions

In addition to handling built-in math (add, sub, mod, etc.) PyLLVM provides a series of intrinsic functions that constitute a simple math library. These functions are: `abs`, `pow`, `exp`, `log`, `sqrt`, `int`, and `float`. These functions are defined in the `mmath` module, and are all functions that LLVM supports although Llvmpy does not provide direct access to them. It is possible to get around this by declaring the function as a header, and LLVM will look up to see if there is a matching function in it's own libraries. While it is possible to directly add the functions to the symbol table, we chose to instead have a dictionary of intrinsic functions that is checked prior to the symbol table when a function call expression is visited. This allows us to have intrinsic math functions that take in and return variable types (i.e. ints or floats).

Print

Print is handled in a similar way to the intrinsic math functions, since Llvmpy does not provide an interface to the `printf` function. However unlike the intrinsic math functions, there is a compiler AST node to handle print.

Branching

Branching, in the form of if statements, while and for loops, are all supported. The only limitation on branching is that new variables cannot be declared within branches - existing ones can be modified. Return statements within if statements are possible only if every branch has a return of that same type - either all branches return or none do. If statements without else statements can do either. Empty lists and vectors, and zero values evaluate to false within the test expression.

Bugs

The only unresolved issue is that of returning newly allocated arrays. Since most of the data lives on the stack, except for returned arrays, the symbol table does not explicitly free anything once it goes out of scope. Currently, the malloc'd arrays that are returned by range are leaked memory. This is not difficult to fix theoretically: add a field in the `Symbol` object to indicate if the variable needs to be freed, and in the `popScope` method of the symbol table, go through and free any malloc'd arrays. Unfortunately there were more critical bugs to fix and there wasn't time for this last one.

3. Backup Method

As PyLLVM does not have full coverage for all possible Python functionality the user could wish to use, a backup method was constructed to avoid full failure in the case that PyLLVM was insufficient. Python provides a Python-C API in which it is possible to wrap Python functions so they are embedded into C++. While it requires more setup than calling C from Python, it is still relatively simple to call Python from C. The primary difference is that in order to send information to Python from C, you need to first convert the C types into Python types. The same goes for receiving information from Python. The Python-C API provides a way to do these conversions. An additional function was added to the Tupleware frontend, in which instead of expecting LLVM code, it takes in the name of the UDF file that will be embedded in C++. Since the conversion from user code to LLVM code need only occur once, when the user is initializing their workflow, even in the worst case (i.e. PyLLVM fails) the efficiency of the workflow is essentially unaffected.

V Related Work

Related Work to Tupleware

Tupleware focuses on high performance, in-memory processing. The most well known analytical framework for cloud-based infrastructure, Hadoop, focuses on data-sets larger than the available memory. Another framework designed for cloud based architectures, Spark, is also an in-memory analytical framework however it uses an iterator model and is not able to explore UDFs. Spark is also restricted to JVM languages, as languages other than Java and Scala require expensive memory copies and JNI/Sockets in order to be integrated. There are also existing analytical frameworks for single machines, such as OptiML, Phoenix, or the BID Data Suite. These systems focus on a single node setup, so unlike Tupleware, they do not optimize an entire workflow into a single distributed program.

Related Work to PyLLVM

While there are similar projects to Tupleware, the focus of this project is on the Python language integration. The project which shares the most important characteristics with PyLLVM is Numba, the JIT specializing Python compiler by Continuum Analytics. However the primary purpose of Numba is not to

generate LLVM IR code, it is to compile Python functions to executables using LLVM as a backend, and call them from Python using the Python-C API. The goal of Numba is to get Python code to run equivalently fast as C, and generating LLVM IR is only a step along the way to native executables.

The architecture of Numba can be described as a six step process, which moves the user's code through five intermediate stages ending with an executable. The first step is generating *Python AST*, which is the same AST representation used by this project (handled by the Python compiler package). The next stage is *normalized IR*: top-level module containers generated by the Python parser are abandoned, and structures like comparisons, list comprehensions, and assignments are normalized. The third stage is *untyped IR in SSA*: control flow is expanded, phi nodes injected, and closures converted so that the intermediate representation is in SSA form. The fourth is *typed IR in SSA*: essentially the same as untyped except contains annotated type information for variables and functions. The fifth stage is *low-level portable IR:* low-level representation that has polymorphic operators. This can be LLVM IR that contains abstract or opaque types. The final product is *final LLVM IR*, which is LLVM assembly code. While the multiple passes and stages that Numba goes through provides more comprehensive coverage of native Python functionality, PyLLVM has the advantage of converting the Python AST representation directly into LLVM IR. Since Numba requires such extensive steps before producing LLVM, it is as lazy as possible and will not compile code that is not executed.

Similar to PyLLVM, the approach taken in Numba is using *type inference* to generate *type information* for the code, so that it is possible to translate Python into statically typed native code. A major advantage of Numba is the ability to both explicitly declare types or infer them. For passing lists as arguments to functions, currently PyLLVM cannot implicitly infer the dimensions. In Numba, the autojit will infer the types of both arguments and return types. Additionally, it is possible to provide Numba with the type-signature of a function explicitly. This will speed up compilation because Numba doesn't have to infer types if the user is able to pre-declare the function signature. PyLLVM does not handle pre-declaring return types in function definitions, so two passes are required to extract the return types from function definitions. Numba is also less limited by static reassignment than PyLLVM. In Numba, variables must have unifying type at control flow merge points (i.e. cannot assign a variable different types in different branches) but can otherwise be reassigned with different types.

A major disadvantage of Numba is that arrays cannot be returned or created (unless in object mode, which is equivalent to the Python-C API backup method in PyLLVM). Numba does not handle returning within a loop, and the user cannot modify a variable defined outside loop within a loop. This is a relatively big disadvantage for the user, since the user may need to allocate and return new arrays without taking a large performance penalty

Numba has explicit support for calling C functions from Python, and PyLLVM does not. This would not be difficult to implement for PyLLVM (in fact, it is what is already happening with the intrinsic math functions, as well as printf). Yet calling C functions from Python does not really fit our requirements, so it is left unimplemented.

In the case where Numba is unable to compile the user's code directly into LLVM, it has a backup method similar to PyLLVM called *object mode*. If Numba can't infer the type of a variable or the user's code contains functionality that is not supported, it uses Python objects. In this mode, Numba generates C code using the Python runtime. While this mode is slower than direct LLVM IR (since the Python code must be compiled to C, and then called from Python), it is still faster than running Python itself, and is equivalently efficient to the PyLLVM backup handling. There are cases where functionality

supported by PyLLVM (without defaulting to the Python-C API) is only supported by Numba in object-mode.

While Numba provides more comprehensive coverage of Python functionality, this additional functionality does not appear important for our purposes. The cases where the user needs decorators or generators, but doesn't need to return or construct array types seems limited. Numba also only provides LLVM IR through command-line tools, either by passing `numba` the argument `--dump-llvm` or by using static compilation (`pycc`). As far as we could tell, there is no way to extract LLVM IR from Numba within a Python program.

Lastly, the goal of this thesis is to investigate the possibility of integrating high-level languages with Tupleware; the majority of compilation techniques used by PyLLVM are language agnostic, while Numba relies on decorators and other Python-specific functionality. There is the additional advantage that PyLLVM, since it was built "in-house", can be easily modified and extended to fit Tupleware's specific needs. There are quite a few other LLVM compilers with similar characteristics, but in this thesis we chose to focus on Python implementations.

VI Analysis

We chose to focus on two specific criteria for our analysis: the usability of the frontend, and code efficiency. We chose four sample algorithms to examine: naive Bayes, k-means, linear regression, and logical regression. It is important to note that while we looked at the speed of compilation, the user's queries are only compiled into llvm *once*, and introduce a very small overhead to the entire process. Even in the worst case, the amount of time spent in compilation would be negligible compared to the cost of analyzing the workflow and deploying the UDFs on the full framework. Additionally, since the LLVM IR generated and passed into the Tupleware framework is expected to be *unoptimized*, the code will be later subjected to the LLVM optimization passes before it is converted to native executables and deployed on the workers. These optimization passes are highly effective and are likely to minimize the differences between the LLVM generated by various compilers. The LLVM IR we are benchmarking in the following section is the unoptimized version, which is not the final code that will be deployed on the clusters.

Benchmarking

PyLLVM

The Python `timeit` module provides a simple way to time small bits of Python code. Benchmarking PyLLVM is relative simple, since it can be called from a Python tester function that is passed into timeit. We chose to compare against Numba for compilation time, which was slightly more complicated because Numba is a lazy compiler and will not compile everything unless it is explicitly used. Numba 0.13 was used to run the sample algorithms, which required a few surface-level syntactical changes to deal with the differences between the syntax of Python 2 and 3. It is important to note that the version of Llvmpy used by this release of Numba is 0.12.3, while the version of Llvmpy we used was 0.12.0. Both versions of Llvmpy are based on LLVM 3.3 and Python 2.7.

We decided to test runtimes for two different types of Python snippets: the first set compiles both the UDF and a test function that generates data and runs the UDF on that data; the second set compiles only the UDF and the data generation is handled externally. The first set is required to ensure Numba will go through the full compilation stages. The second set is most similar to how the compilation would occur within the Tupleware framework. However in order to call the first set of sample algorithms, data needs to be loaded into memory within the code that will be compiled to LLVM. Loading data requires the creation of arrays, which Numba cannot handle without defaulting to object mode (using the Python runtime to compile the function definitions, which is inherently slower). The initial versions of these sample functions had arrays explicitly declared and returned by the UDFs, which were changed to be passed as arguments so not to force Numba to use object mode. Unfortunately there is no way to avoid entering object mode when reading in data in Numba. Each algorithm was run through `timeit` for three trials, each with 500 iterations on a single AWS m1.small instance. The results of the first set of tests, where the UDF and tester functions were compiled, can be found in Figure 6.

Benchmarks for UDF+Test Function (Set 1)

Algorithm	Naive Bayes	K-means	Linear Regression	Logical Regression
PyLLVM				
Total runtime (seconds)	303.326012135	703.023230076	430.95607686	441.539062977
Average per cycle (seconds)	0.60665202427	1.406046460152	0.86191215372	0.883078125954
Numba				
Total runtime (seconds)	205.806375027	301.936852217	209.830909014	201.937881947
Average per cycle (seconds)	0.41161275006	0.60387370444	0.419661818028	0.403875763894

Figure 6: Benchmarks for Test Set 1

Benchmarks for UDF only (Set 2)

Algorithm	Naive Bayes	K-means	Linear Regression	Logical Regression
PyLLVM				
Total runtime (seconds)	81.2685310841	163.382500887	83.3339309692	97.6413960457
Average per cycle (seconds)	0.162537062169	0.32676500178	0.1666678619384	0.1952827920914
Numba				
Total runtime (seconds)	174.092276096	892.465705872	220.414202929	323.593291998
Average per cycle (seconds)	0.348184552192	1.78493141175	0.440828405858	0.647186583996

Figure 7: Benchmarks for Test Set 2

As can be seen from Figures 6, Numba is consistently around twice as fast as PyLLVM for the first set of benchmarks. There are a variety of reasons for these results, but the primary discrepancy most likely comes from the fact that generating data in PyLLVM is more expensive than Numba. Numba has to capacity to use Numpy arrays which are very fast, while PyLLVM goes through and populates each array by hand. Numpy is a scientific computing package for Python that provides an array type that is, for large data, faster and more compact that Python lists. The time to it takes compile data generation is larger than the cost of compiling the UDF, which can be seen in the second set of results found in Figure 7. However in order for Numba to call any external math functions, like sqrt or exp (found in k-means and logical regression), it needs to import the Math module. PyLLVM has intrinsic support for these functions, and

does not require importing. This could explain the spike in runtime for the k-means algorithm with Numba as it requires more complicated math functions than the other tests.

The second set of results, where the data generation is handled by Python and not compiled to LLVM, has PyLLVM performing faster than Numba. However it is hard to compare PyLLVM and Numba in this case because PyLLVM will compile only the UDF, while in order for Numba to fully compile just the UDF, it needs to be called to force Numba to go through the full compilation process. Numba has the capacity to compile a function to LLVM, and then call the executable LLVM from Python. PyLLVM does not have this functionality, as it is not intended to be a tool for speeding up Python execution. In order to call the LLVM generated by PyLLVM, the LLVM `lli` tool needs to be called externally.

Generated LLVM

We chose to benchmark the generated LLVM to get an idea of how the LLVM IR code generated by PyLLVM compares with unoptimized LLVM IR generated by other compilers, such as Clang. We ran the LLVM IR code using `lli`, a tool provided by the LLVM framework to directly execute programs from LLVM bitcode. There are not many programs dedicated to benchmarking unoptimized LLVM bitcode, so we used linux system call `time` to analyze these runtimes. Although there are issues with using system time for benchmarking, our goal was primarily to determine if there was some great discrepancy between our LLVM and the LLVM generated by Clang. The results can be found in Figure 8.

Benchmarking LLVM

Algorithm	Naive Bayes	K-means	Linear Regression	Logical Regression
PyLLVM				
real	3m39.456s	4m48.931s	4m5.386s	4m20.288s
user	1m48.487s	2m50.827s	2m13.984s	2m27.721s
sys	1m46.095s	1m52.703s	1m45.707s	1m47.363s
Clang				
real	2m59.670	3m18.416	3m0.939s	3m5.402s
user	1m15.261	1m35.330	1m19.257s	1m23.133s
sys	1m37.050	1m38.246	1m35.230s	1m36.854s

Figure 8: Benchmarks for Generated LLVM

The time difference between the LLVM generated by PyLLVM and Clang is not large enough to have much of an impact on the user. We are focusing on system time, so there is consistently less than a minute difference between the runtimes. The difference between runtimes for the system time is as follows: 1% for Naive Bayes, 12% for K-means, 9% for Linear Regression and 9% for Logical Regression. A potential explanation for the spike in K-means is that the way PyLLVM handles square root and other intrinsic math functions is by calling a separate function that calls the built-in square root function. This is because Llvmpy does not provide an interface to call sqrt directly. This means that for every `sqrt` call, PyLLVM has to go through 2 function layers while Clang will only require 1.

In fact, considering that Clang is a well established tool that has been part of the LLVM release cycle since LLVM 2.6 and aims to replace GCC, PyLLVM performs well in comparison. This is primarily due to the optimization passes that most likely squash the surface differences between the codes before they are run.

Usability

The general criteria we used to determine just how successful PyLLVM is at providing a Python subset that meets the needs of Tupleware users is (1) how different it is from the existing C++ frontend (2) how close to real Python it is.

As can be seen from the examples, the difference between in the sample algorithms in Python and C++ is not large. Since machine learning algorithms tend to be short and use simple mathematical functionality, it is not expected that converting the simplest C++ functions to Python would have largely different code lengths. The primary advantage of Python over C++ is the user is not forced to deal with their own memory management and other bookkeeping. If the user does not have to worry about pointers and arguments going out of scope, they are free to focus on their algorithms and workflow. Additionally, within the machine learning community many algorithms are already implemented and supported in Python. Lastly, frameworks competing with Tupleware generally have frontends in languages that are high-level than C++. Ultimately, PyLLVM is successful in providing an interface for users that does not lose any usability from the existing C++ frontend.

The subset of Python handled by PyLLVM is unquestionably more limited than Python itself. However if we focus on the user's needs - i.e. simple mathematical algorithms, the loss of features like objects, decorators, and generators is a not major limitation. However the fact that lists are not dynamically resizable is limiting, although not more so than the existing C++ frontend. However part of the attraction of Python is that it is dynamically typed, so the fact that variables are not dynamically reassignable does restrict the user. Additionally, unlike regular Python the user needs to declare the dimensions of lists they are passing as function arguments. This can be a major weakness if the user isn't able or doesn't want to hard-code the lengths of arrays into their UDFs. Lastly, n-dimensional arrays are not available in PyLLVM although that is more a matter of implementation than it not being possible.

Overall, the Python tupleware frontend is simpler to use than the existing C++, but provides a

limited subset of Python to the user. Depending on what the user requires, it may be better for them to use Numpy for algorithms that require fancier functionality. However with complicated functionality, Numpy defers to the Python runtime so the cost of compilation will be comparable to that of PyLLVM. The subset of Python provided by PyLLVM is more limiting functionality-wise than the existing C++ frontend, for the reason that the C++ frontend can handle the entire C++ language while PyLLVM is only a subset of Python. Ultimately, for the specific algorithms analyzed in this section, PyLLVM does not impose major limitations on the user however there are many cases that it would.

VII Conclusion and Future Work

The goal of this thesis was to tightly integrate a higher-ordered language, such as Python, into a general purpose distributed analytical framework, like Tupleware, without sacrificing the usability of the language. The integration of Tupleware to Python was successful in that the Tupleware frontend is fully accessible from Python. From the user's point of view, the entire Python language is supported - however not without the performance hits of using the Python-C API as a backup method.

The future goals of this project are to expand PyLLVM so that fewer cases need to be handled by the backup method. While the core Python language is supported, dynamically typed variables will hopefully be added soon. Additionally, support for multidimensional lists can be easily added. The next steps for PyLLVM beyond dynamic types is introducing more native data types, especially a dictionary type. While there could be demand for more complicated functionality like objects, dynamic scoping, generators, etc. the implementation would require multiple passes by the compiler. The goal of this project is to provide a simple, easy to understand and easy to extend compiler for Python. With more functionality comes more complex code, and it is important not to lose the simplicity of design found in the current implementation.