PyLLVM
A compiler from a subset of Python to LLVM-IR

Anna Herlihy
MongoDB
EuroPython Bilbao 2016
Outline

1. Motivation
2. PyLLVM Features
3. Related Work
4. Analysis and Benchmarking
5. Conclusion
Motivation
Motivation: Tupleware

- Distributed analytical framework built at Brown for running algorithms on large datasets
- User supplies:
  1. data
  2. UDF (algorithm)
  3. workflow (map, reduce, join, etc.)
- Goal: language and platform independence
Motivation: The LLVM Compiler Infrastructure Project

- LLVM-IR is a transportable intermediate representation by the LLVM Compiler Project

Languages:
- Julia
- R
- C
- Python
- Go

Platforms:
- x86/x86-64
- AMD
- ARM

(And more)
The goal of this project is to provide a Python interface with Tupleware’s C++ backend to make the user experience as simple and straightforward as possible.
Mission: Python and Tupleware

Workflow
- map, filter, reduce, combine, join, loop, etc.

Algorithm
- k-means, Naive Bayes, linear regression, etc.

This talk

Tupleware
- C++ Frontend Operators

Operators
- Boost Python
- PyLLVM

C++

Python
from TupleWare import load

def linreg(dims, data, w):
    dot = 1.0
    c = 0
    while c < dims:
        dot += data[c]*w[c]
        c += 1
    label = data[dims]
    dot *= -label
    c2 = 0
    while(c2 < dims):
        g[c2] += dot*data[c2]
        c2 += 1

def run_map(data):
    TS = load(data)
    TS.map(linreg)
    TS.execute()
import PyLLVM
import TupleWrapper  # Boost C++ binding

def map(self, udf):
    try:
        # Try to get LLVM-IR from PyLLVM.
        llvm = PyLLVM.compiler(udf)
    except PyLLVM.PyllvmError:
        # Unable to compile the UDF, try backup.
        self.backup_map(udf)
    except Exception as exc:
        # The exception was semantic.
        raise ValueError("Bad Python in UDF", exc)
    else:
        # Valid LLVM IR was generated
        # can now call desired operator
        TupleWrapper.map(llvm)
PYLLVM
PyLLVM

- 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.

- Based on py2llvm, an unfinished Google Code project from 2010
  - [https://code.google.com/p/py2llvm/](https://code.google.com/p/py2llvm/)

- Uses llvmpy: wrapper for C++ IR Builder
PyLLVM: Subset of Python

- Anticipated common requirements for Tupleware users:
  - Machine learning algorithms are often simple, easily optimized mathematical functions
- Primarily statically type-inferable code is handled
- No dictionaries, list comprehensions, or objects.
PyLLVM: Overview of Design

- **Abstract Syntax Tree:**
  - Python 2.7’s compiler package: parse, walk

- **Semantic analysis**
  - CodeGenLLVM: Visitor class
    - SymbolTable: Keeps track of variables and scope
    - TypeInference: Infers expression type

- **Code Generation**
  - llvmpy: Generates LLVM-IR: Python bindings to the C++ LLVM IR-Builder
Static Single Assignment

- LLVM instructions are SSA: Registers can only be assigned to once

- Result of being halfway between programming language and machine code

- Do not want to implement entire compiler in SSA form...
Scoping and Variables

SOLUTION: variables are allocated on the stack and addresses stored in SymbolTable

- Symbol: class representing variable
  - name, type, memory location, etc.

- SymbolTable: stack of tuples, each representing a scope
  - Scope contains name and map of varname to Symbols
LLVM Types
Types: PyLLVM

LLVM IR Types: Integers, floats, pointers, arrays, vectors, structs, functions

PyLLVM Types: integers, floats, vectors, lists, strings, functions
Inferring Types

- LLVM-IR is statically typed, Python is not
- TypeInference infers Python types from nodes of the AST
  - recursively traverses tree until reaches leaf node, infers based on leaf
  - uses symbol table for variables/functions
- Intrinsic math functions return the type they are passed in to avoid multiple functions for integer vs. float
PyLLVM Types

1. Numerical Values
2. Vectors
3. Lists
4. Strings
5. Functions
6. Branching and Loops
Numerical Values

- Integers
  - LLVM 32-bit integers
- Floats
  - LLVM 32-bit floating point
- Booleans
  - 1-bit integers
    - converted to 32-bit before being stored
  - $True + True = 2$
PyLLVM Types

1. Numerical Values
2. Vectors
3. Lists
4. Strings
5. Functions
6. Branching and Loops
Vectors

- 4-element immutable floating point vector types
  - `vec = vector(1, 2, 3, 4)`
  - `vec.x/y/z/w` or `vec[i]`

- Built in: add, subtract, multiply, divide, compare

- Written specifically for ML functions
PyLLVM Types

1. Numerical Values
2. Vectors
3. **Lists**
4. Strings
5. Functions
6. Branching and Loops
Lists (WIP)

- Static-length mutable lists
  - range, zeros, len
- Based on underlying LLVM array type
  - can be populated with constants or pointers
- \texttt{alloca_array’d} onto stack and passed by pointer (unlike vectors)
  - \textit{Any lists returned from functions will be stored on the heap}
PyLLVM Types

1. Numerical Values
2. Vectors
3. Lists
4. **Strings**
5. Functions
6. Branching and Loops
Strings

- Desugared into lists of integers
  - strings are lists of characters
  - characters can be represented as integers

- Symbol table remembers if list variable contains integers or characters
  - For print, cmp, etc

- That was easy!
PyLLVM Types

1. Numerical Values
2. Vectors
3. Lists
4. Strings
5. **Functions**
6. Branching and Loops
Function Definitions

- Can define and call functions from anywhere in the UDF
- Function signature generated and arguments added to the symbol table
- The only time where the compiler does 2 passes:
  - One descent to extract return type of func
  - Pops symbol table scope, calls delete on LLVM-IR Builder, and runs pass again
Function Arguments

- Since types are not dynamic, all arguments must have type values
  - `func(i=int, f=float)`
- Type and length of list must be specified
  - `func(l=listi8)`
  - *ONLY* place where subset of Python differs from real Python
- Can be implemented in future, if only PEP484 (Type Hints) had been reality...
Intrinsic Functions

- Simple built-in math library
  - `abs`, `pw`, `exp`, `log`, `sqrt`, `int`, `float`
  - takes in variable type, returns same type
- `llvmpy` does not provide access to equivalent IR instruction
  - Workaround: declare function as header, LLVM-IR will look up matching function
- print
  - handled similarly to intrinsic math functions
PyLLVM Types

1. Numerical Values
2. Vectors
3. Lists
4. Strings
5. Functions
6. Branching and Loops
Conditionals: if, for, while

- All supported with some limitations:
  - new variables declared within branches will go out of scope upon exit
  - existing vars can be modified
  - return within if statements supported only if every branch contains return

- All types have boolean values
  - empty lists are false, nonzero values are true
Related Work
Numba

- JIT specializing Python compiler by Continuum Analytics

- Purpose is to compile functions into executables using LLVM and call them from Python using the Python-C API

- **Goal is to get Python to run fast, generating IR is only a step along the way**
PyLLVM and Numba Comparison

- **Bottom line: same tools, different goals**
- Numba provides comprehensive coverage of Python, and is a more mature project
- In order get LLVM-IR out of Numba, have to run `numba --dump-llvm` or use `pycc`
- PyLLVM build “in-house”
Analysis

• Focused on two specific criteria for analysis
  ○ Usability of the frontend
  ○ Code efficiency
  ○ Difficult to compare compilation time

• Sample algorithms: Naive Bayes, k-means, linear regression, and logical regression.
Analysis: Usability

- PyLLVM does not lose any usability
- Primary advantage of Python is freedom from memory management and other bookkeeping

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

```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]];
}
```
Analysis: Benchmarking

- **Compilation:** PyLLVM vs. Numba
  - Only happens once, cost is minor

- **Generated LLVM:** PyLLVM vs. Clang
  - Tested unoptimized LLVM, ultimately differences likely to be optimized away
Analysis: Executable Runtime

- Generated unoptimized LLVM-IR using clang
- Ran generated LLVM-IR using lli
- Used system time to compare runtime
- Ran algorithm 2500 times, for 500 trials
Analysis: Executable Runtime

![Generated LLVM-IR Chart]

- **Systems time (ms)**
  - Naive Bayes: Pylivm 100, Clang 100
  - Kmeans: Pylivm 120, Clang 120
  - LinReg: Pylivm 110, Clang 110
  - LogReg: Pylivm 110, Clang 110
Results

● Difference between runtimes for system time is:
  ○ Naive Bayes: 1%
  ○ K-means: 12%
  ○ Linear regression: 9%
  ○ Logical regression: 9%

● Spike in k-means potentially because sqrt
  ○ llvmpy does not provide direct access to LLVM’s sqrt instruction
Conclusion

● Overall, were able to achieve goal
  ○ Able to fully integrate Python as a Tupleware frontend
  ○ To the user, all of Python is supported (although with performance hit)

● Future work: Dynamically typed variables, dynamic-length and multidimensional lists, new native data types (dicts!)
Acknowledgements

- Thank you to Tim Kraska my advisor!
- Alex Galakatos, Andrew Crotty and Kayhan Dursun for Tupleware help
- Thank you to the lost souls who wrote py2llvm
- Thank you to MongoDB, specifically A. Jesse Jiryu Davis and Bernie Hackett for encouraging me to talk

- Thank you EuroPython!
herlihyap@gmail.com

Original: code.google.com/p/py2llvm

My work: github.com/aherlihy/PythonLLVM

Tupleware: tupleware.cs.brown.edu

Twitter: @annaisworking