LPython: Novel, Fast, Retargetable Python Compiler

PyCon India 2023

Ubaid Shaikh, Ondřej Čertík, Brian Beckman, Gagandeep Singh, Thirumalai Shaktivel, Smit Lunagariya, Naman Gera, Pranav Goswami, Rohit Goswami, Dominic Poerio, Akshānsh Bhatt, Virendra Kabra, Anutosh Bhat, Luthfan Lubis, Dylon Thomas

September 30, 2023

Introduction

Talk's agenda

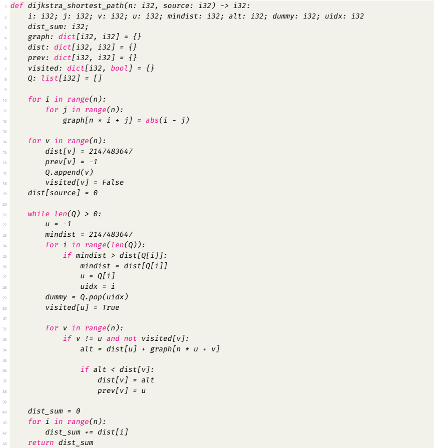
- Can we write a Python compiler that compiles fast and generates fast code?
- Online Demo
- Compiler stuff:
  - AST, ASR, Compiler Backends
  - AOT, JIT, Interoperability with CPython
- Summary

LPython

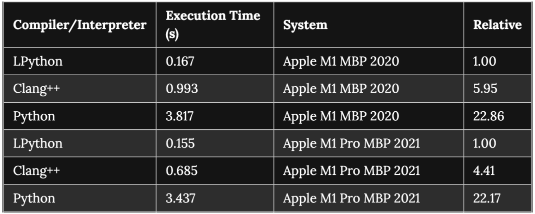
LPython - Motivation

- Python - long favoured for simplicity, intuitive syntax, productivity, ecosystem
- But inherently slow compared to other compiled langauges such as C and C++.
- LPython - Type Annotated Python Compiler for performance
- Most existing Python compilers focus on improving Python performance
- LPython aims to run Python at the maximum possible speed

LPython compared to C++

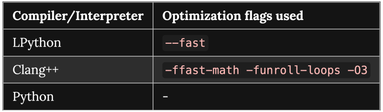


```
def dijkstra_shortest_path(n: i32, source: i32) -> i32:
    i: i32; j: i32; v: i32; u: i32; mindist: i32; alt: i32; dummy: i32; uidx: i32
    dist_sum: i32;
    graph: dict[i32, i32] = {}
    dist: dict[i32, i32] = {}
    prev: dict[i32, i32] = {}
    visited: dict[i32, bool] = {}
    Q: list[i32] = []

    for i in range(n):
        for j in range(n):
            graph[n * i + j] = abs(i - j)

    for v in range(n):
        dist[v] = 2147483647
        prev[v] = -1
        Q.append(v)
        visited[v] = False
    dist[source] = 0

    while len(Q) > 0:
        u = -1
        mindist = 2147483647
        for i in range(len(Q)):
            if mindist > dist[Q[i]]:
                mindist = dist[Q[i]]
                u = Q[i]
                uidx = i
        dummy = Q.pop(uidx)
        visited[u] = True

        for v in range(n):
            if v != u and not visited[v]:
                alt = dist[u] + graph[n * u + v]

                if alt < dist[v]:
                    dist[v] = alt
                    prev[v] = u

    dist_sum = 0
    for i in range(n):
        dist_sum += dist[i]
    return dist_sum
```



Compiler/Interpreter	Execution Time (s)	System	Relative
LPython	0.167	Apple M1 MBP 2020	1.00
Clang++	0.993	Apple M1 MBP 2020	5.95
Python	3.817	Apple M1 MBP 2020	22.86
LPython	0.155	Apple M1 Pro MBP 2021	1.00
Clang++	0.685	Apple M1 Pro MBP 2021	4.41
Python	3.437	Apple M1 Pro MBP 2021	22.17

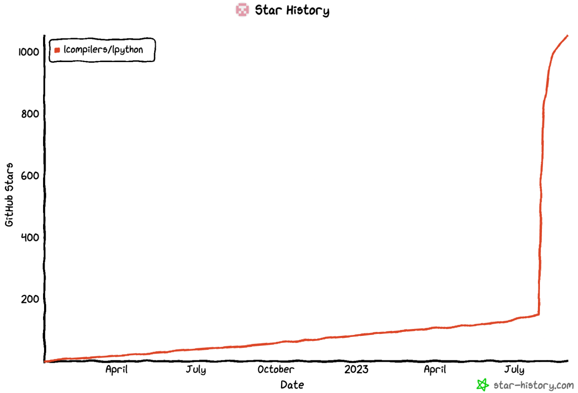
Note that the following optimization flags were used:



Compiler/Interpreter	Optimization flags used
LPython	`--fast`
Clang++	`-ffast-math -funroll-loops -O3`
Python	-

More Benchmarks at https://lpython.org/blog/2023/07/lpython-novel-fast-retargetable-python-compiler/

Star History



Spike in star count when we released LPython alpha

LPython

```python
from lpython import i32
from  numpy import empty, int32

def main0():
    x: i32 # i32 represents int32
    x = (2+3)*5
    print(x)

    a: i32[5] = empty([5], dtype=int32)
    print(a)

if __name__ == "__main__":
   main0()
```

Type annotated example code

- Supports subset of CPython
- Requires types (annotations)
- If LPython compiles and runs, it will run in CPython
- Never slower than C and C++
  - If you find an example where it is slower, it is a bug and we request you to report it to us.
- Several backends: LLVM, C, C++, WASM, X64 (via WASM)

Online Demo (dev.lpython.org)

LPython Annotation Types

LPython - Annotation Types

```python
a: i32
b: i64 # similarly i8, i16
c: u32
d: u64 # similarly for u8, u16
e: f32
f: f64
g: c32
h: c64

i: bool
j: Const[i32] = 5 # similarly other types
k: CPtr
l: i32[5] = empty([5], dtype=int32)
m: Allocatable[i32[:]] = empty([n], dtype=int32)

n: list[f32]
o: tuple[i32, i64, str]

@dataclass
class Student:
    name: str
```

Annotations for variables

- Supports Integer types - *`i8`*, *`i16`*, *`i32`*, *`i64`*
- Similarly unsigned integers
- Floating points and Complex numbers
- Constant variables that need initialization value at the time of declaration
- Aggregate types like *`list`*, *`tuple`*, *`classes`* and more

LPython - Annotation Types

```
def sayHi():
    print("Hi")

def add(a: i32, b: i32) -> i32:
    ...

T = TypeVar('T')
@restriction
def sub(x: T, y: T) -> T:
    ...

def f(a: In[Student], b: InOut[f32[:]]):
    ...

@lpython(backend="c", backend_optimisation_flags=["-ffast-math"])
def g(n: i32):
    ...

@pythoncall(module="email_extractor_util")
def get_email(text: str) -> str:
    ...
```

Annotations, decorators for functions

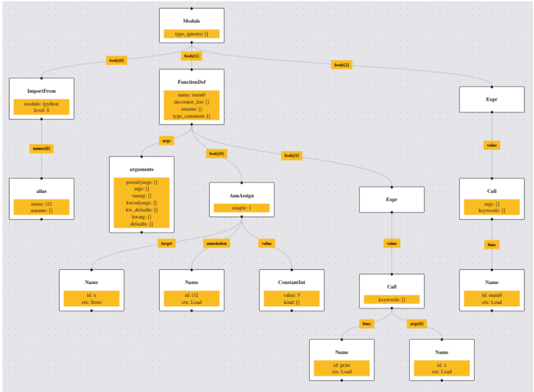
- Annotate function parameters and return types
- Supports generics.
  - We have ongoing work on it.
- Supports specifying intents (In, InOut, Out) for function parameters.
  - `In` - variable cannot be modified
  - `InOut` - variable can be modified
- `@lpython` and `@pythoncall` decorators
  - `lpython` calling from CPython into LPython
  - `pythoncall` calling from LPython into CPython

Abstract Syntax Tree and Abstract Semantic Representation

Abstract Syntax Tree (AST)

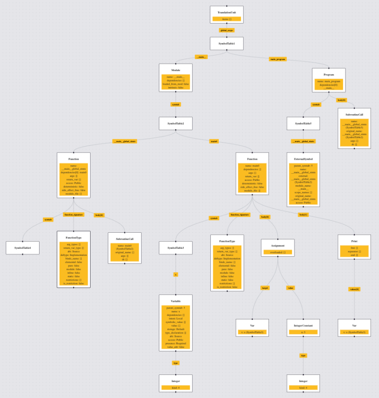
```python
from lpython import i32

def main0():
    x: i32 = 5
    print(x)

main0()
```

Example



AST

Abstract Semantic Representation (ASR)



ASR

- Independent of the frontends and backends
- As high level as possible, but faithful to the surface language
- ASR → ASR passes
  - loop vectorize
  - dead code removal
  - inline function calls, etc.
- ASR → Backends

Backends

Backends

- Can compile Type-annotated Python code to different targets
- Current supported backends
    - LLVM
    - C
    - C++
    - WASM
    - X64 (via WASM)
- Other backends planned (Python, Fortran, etc.)

LPython - Usage

- `lpython main.py --show-tokens`
- `lpython main.py --show-ast`
- `lpython main.py --show-ast --tree`
- `lpython main.py --show-ast --json`
- `lpython main.py --show-ast --visualize` (earlier image)
- `lpython main.py --show-asr`
- `lpython main.py --show-asr --tree`
- `lpython main.py --show-asr --json`
- `lpython main.py --show-asr --visualize` (earlier image)
- `lpython main.py --show-c`
- `lpython main.py --show-cpp`
- `lpython main.py --show-wat`
- `lpython main.py --show-llvm`
- `lpython main.py` (by default LLVM backend is used. Compiles and runs the code)
- `lpython main.py --backend=llvm`
- `lpython main.py --backend=c`
- `lpython main.py --backend=wasm`

Example of compiling to C

```python
from lpython import i32

def add(a: i32, b: i32) -> i32:
    return a + b

def main0():
    x: i32 = 5
    y: i32 = 3
    print(x, y)

    print(add(x, y))

main0()
```

main.py

```
$ lpython main.py --show-c > main.c
```

Example of compiling to C

```c
#include <inttypes.h>

#include <stdlib.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <lfortran_intrinsics.h>

int32_t add(int32_t a, int32_t b);
void main0();
void __main__global_stmts();



// Implementations
int32_t add(int32_t a, int32_t b)
{
    int32_t _lpython_return_variable;
    _lpython_return_variable = a + b;
    return _lpython_return_variable;
}
```

```c
void main0()
{
    int32_t x;
    int32_t y;
    x = 5;
    y = 3;
    printf("%d%s%d\n", x, " ", y);
    printf("%d\n", add(x, y));
}

void __main__global_stmts()
{
    main0();
}

int main(int argc, char* argv[])
{
    _lpython_set_argv(argc, argv);
    __main__global_stmts();
    return 0;
}
```

Ahead of Time, Just In Time and CPython Interoperability

Ahead of Time and Just In Time Compilation

AOT

- By default compiles to LLVM when no backend is specified
- Other backends can be used by using `--backend=c`
- Supports C, C++, WASM, WASM_X64 (which generates a lean binary)

JIT

- Just decorate Python function with `@lpython`
- Specifying the desired backend as, `@lpython(backend=“c”)` or `@lpython(backend=“llvm”)`
- Supports C Backend at the moment, LLVM and others planned

Interoperability with CPython

email_extractor.py (LPython)

```python
# get_email is implemented in email_extractor_util.py
# which is intimated to LPython by specifiying
# the file as module in `@pythoncall` decorator

@pythoncall(module="email_extractor_util")
def get_email(text: str) -> str:
    pass

text: str = "Hello, my email id is lpython@lcompilers.org."
print(get_email(text))
```

email_extractor_util.py (CPython)

```python
# Implement `get_email` using `re` CPython library

def get_email(text):
    import re
    # Regular expression patterns
    email_pattern = r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b"

    # Matching email addresses
    email_matches = re.findall(email_pattern, text)

    return email_matches[0]
```

- Supports C backend currently

```
% lpython email_extractor.py --backend=c --enable-cpython
lpython@lcompilers.org
```

Summary and Future Work

Summary

The compiler itself

- Implemented in C++
- Compiles in 30s
- Fast compilation
- Fast runtime
- Custom AST/ASR tree representation and visitors
- Quickly loads and runs in a webpage
- https://dev.lpython.org/

Future Work

- More complete NumPy support (arrays)
- Strong support for structs and pointers (allows general programming)
- Make sure LPython is at least as fast as C++ for every simple benchmark (currently we have a few where we are slower)
- Make ‘@pythoncall‘ work with the LLVM backend
- Allow to extend LPython with custom hardware backends (GPU, APU, ...)
- Use LPython for bigger projects (ML, compilers, etc.)
- Optimizations

Thank You!

If you’re excited about the work we’re doing on LPython, we invite you to come and contribute. We’re always looking for new contributors to help us improve the project and make it more useful for everyone.

Contact: shaikhubaid769@gmail.com

GitHub: github.com/Shaikh-Ubaid