Distributed Computing and Introduction to High Performance Computing

Ahmed Ratnani[1]
[1]Mohammed VI Polytechnic University, Benguerir, Morocco

Outline of this lecture

- Some basics on Programming Languages and Compilers
- What is Numba?
- What is Pyccel?
- ➠ *see the last Pyccel talk*

A brief history on programming languages

- Lambda Calculus (A. Church - 1930s): formal system for expressing computation based on functions
- Turing machine (1936): Abstract machine for computing (abstract) functions
- Von Neumann model (1945): first model for computers
- First high-level language:
  - Fortran (J. Backus - 1954)
  - Lisp (J. Mc Carthy - 1958)
- Two paradigms emerged:
  - imperative programming
  - functional programming

Language Theory

A brief introduction to Compilers

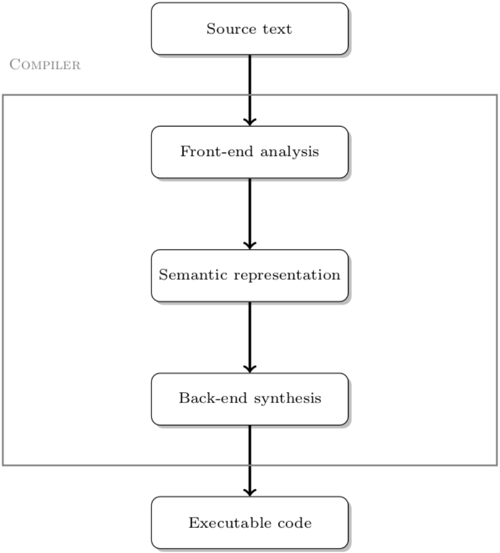


Figure 1: Conceptual structure of a compiler

Language Theory

A brief introduction to Compilers

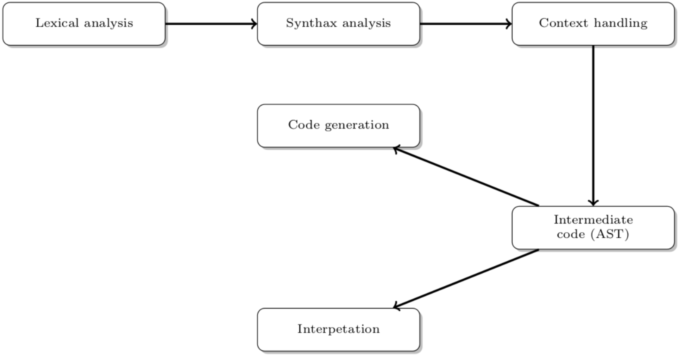


Figure 2: Conceptual structure of a compiler

Numba

What is Numba?

The translation/magic is been done using the LLVM compiler, which is open sourced and has quite active dev community.

- Numba compiles Python code at runtime to native machine instructions
- Unlike Cython, it is backward compatible (same for Pyccel and Pythran)
- It is based on the LLVM compiler
- Open source code with a big active community

Installation

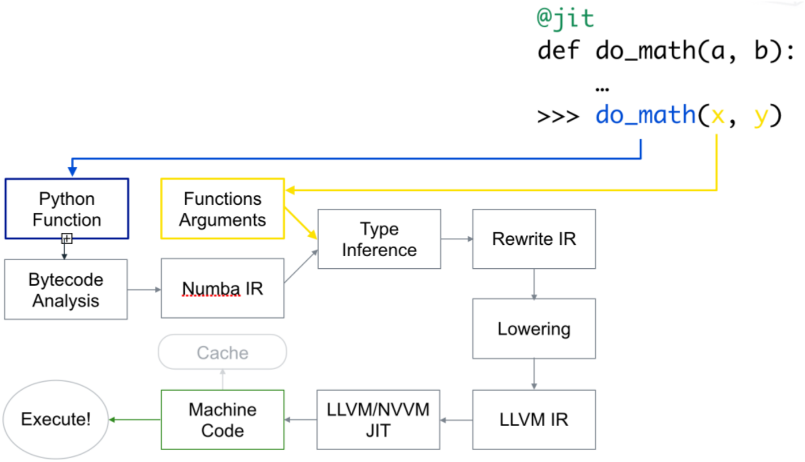
```
pip install numba
```

Typical usage

```
@jit
def myfunction(*args, **kwargs):
  ...
```

Numba

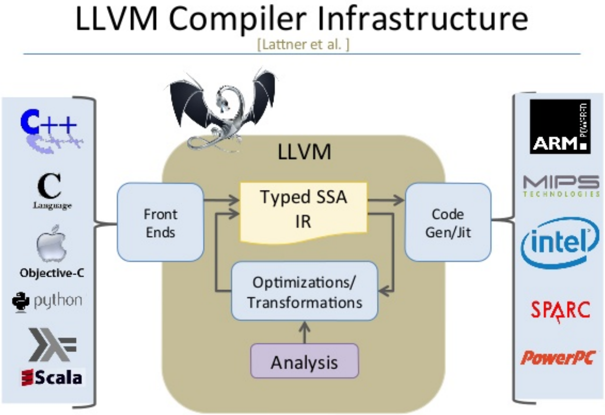
Workflow



Numba

Current limitations

- Numba compiles Python functions, not entire programs (pypy is great for that). It also doesn't support partial compilation of functions – it needs to be able to resolve all data types in the selected function.
- Presently, Numba is focused on numerical data types, like int, float, and complex. There is very limited string processing support and the best results are realised with Numpy arrays.
- Decorating functions that make use of Pandas (or other unsupported data structures) would deteriorate performance. Pandas is not understood by Numba and as a result, Numba would simply run this code via the interpreter but with the additional cost of the Numba internal overheads.
- You are better off using Cython for code that interferes with C++, as Numba can't talk with C++ effectively unless a C wrapper is used.
- Numba doesn't generate C/C++ code that can be used for a separate compilation; it goes directly from Python down to LLVM code. Cython would be more suitable for this use case, as it allows inspection of the code in C++ before compilation.


LLVM Compiler Infrastructure
[Lattner et al.]
C++
C
Language
Objective-C
python
Scala
LLVM
Front
Ends
Typed SSA
IR
Code
Gen/Jit
Optimizations/
Transformations
Analysis
ARM
MIPS
TECHNOLOGIES
intel
SPARC
PowerPC

LLVM

Compiler system

- The LLVM Compiler Infrastructure
  - Provides reusable components for building compilers
  - Reduce the time/cost to build a new compiler
  - Build static compilers, JITs, trace-based optimizers, ...
- The LLVM Compiler Framework
  - End-to-end compilers using the LLVM infrastructure
  - C and C++ gcc frontend
  - Backends for C, X86, Sparc, PowerPC, Alpha, Arm, Thumb, IA-64, ...

- The LLVM Virtual Instruction Set
  - The common language- and target-independent IR
  - Internal (IR) and external (persistent) representation
- A collection of well-integrated libraries
  - Analyses, optimizations, code generators, JIT compiler, garbage collection support, profiling, . . .
- A collection of tools built from the libraries
  - Assemblers, automatic debugger, linker, code generator, compiler driver, modular optimizer, . . .

Goals of the Compiler

- Analyze and optimize as early as possible
  - Compile-time opts reduce modify-rebuild-execute cycle
  - Compile-time optimizations reduce work at link-time (by shrinking the program)
- One IR (without lowering) for analysis and optimization
  - Compile-time optimizations can be run at link-time too
  - The same IR is used as input to the JIT

LLVM

Goals of the IR

- Easy to produce, understand, and define
- Language- and Target-Independent
  - AST-level IR is not very feasible
  - Every analysis/transformation must know about all languages
- One IR for analysis and optimization
  - R must be able to support aggressive IPO, loop opts, scalar opts, ... high- and low-level optimization
- Optimize as much as early as possible
  - Can't postpone everything until link or runtime
  - No lowering in the IR