Speeding up Python code with C/C++

Dmitri Rozmanov

WestGrid HPC summer school at UofC 2019

Information:

- **Coffee break**: 14:30 -- 14:45 in the Atrium.

- Shared **Google Doc**: http://bit.ly/UofCPythonCpp

- **Logging** to cluster:

```
$ ssh username@arc.ucalgary.ca
```

- **Interactive** node request command:

```
$ salloc -t 4:00:00 -N 1 -n 1 -c 1 --mem=1gb
                    -p single --reservation=ss2019
```

Introduction

- Python is a very **nice** language.
- Python has lots of **libraries**.
- Python is used very **widely**.
- Python library functions usually run very **fast**.

- Pure Python is very **slow** (interpreted language).
- To write fast code you have to **wrap you ideas** into the language of the library.
- What do you do if there is **no library** that does what you need**?**

- Flexibility of using Python and still being able to do exactly what you want, **quickly**.

Today we will

- Learn about several way to **call C/C++ code** from Python.
- Pick **one method** and look at it in more details.
- Find the most **time-consuming part** of a test case Python code.
- Implement the **time-critical part in C++**.
- **Modify the Python program** to take advantage of the compiled C++ function.
- Evaluate the **speed-up**.
- Make conclusions.

Development environment

- **Bash** command line shell:

```
$ ssh username@arc.ucalgary.ca
```

- **Python 2.7** and **GCC 4.8.5 (default)**:

```
$ module load python/anaconda2-2018.12
```

- **GNU Screen** shell session manager:

```
$ screen
```

- **Text editor**: vi, **vim**, nano, **mcedit**, emacs.

```
$ vim my_code.py
$ mcedit my_code.py
```

GNU Screen

Screen is a full-screen window manager for several interactive shells.

- Quick **switching** between the script editing and running windows;
- Session will **persist** if you close the lid of your laptop.
  But you have to reconnect.

Minimal useful **commands**:

`$ man screen`

- Create a new window: **C-a c**
- Close a window: **C-d**
- Detach screen from the this terminal: **C-a d**
- Switch to another window: **C-a “**
- Toggle between two recent windows: **C-a C-a**
- Reconnect: `$ screen -r`

How to call C / C++ functions from Python?

- **Python-C-API** is the backbone of the standard Python interpreter, **CPython**. Using this API it is possible to write Python extension module in C and C++.

- **CTypes** is included in **Python 2.5** and later. CTypes lets you talk directly to shared libraries on both Windows and UNIX.

- **SWIG**: **Simple Wrapper Interface Generator**. SWIG is capable of wrapping C in a **large variety of languages**.

- **Cython** is both a **python-like** language for writing C-extensions and an advanced compiler for this language.

How to call C / C++ functions from Python?

- **Pyrex** is a **Python-like** language used to create C modules for Python.

- **SIP** is used to generate Python **bindings for Qt** (PyQt), a graphics library. It can be used to wrap any C or C++ API.

- **Boost.Python** lets you run **C++ code** from Python, and Python code from C++, seamlessly.

- Resources:
  - SciPy lecture: http://www.scipy-lectures.org/advanced/interfacing_with_c/interfacing_with_c.html
  - Software carpenty: http://intermediate-and-advanced-software-carpentry.readthedocs.io/en/latest/c++-wrapping.html

How to call C / C++ functions from Python? Cont.

	Part of Python	Compiled	Autogenerated	Numpy Support
Python-C-API	Yes	Yes	No	Yes
CTypes	**Yes**	**No**	**No**	**Yes**
SWIG	No	Yes	Yes	Yes
Cython	No	Yes	Yes	Yes
pyrex	No	Yes	Yes	?
SIP	No	Yes	Yes	?
Boost.Python	No	Yes	?	?

CTypes

is a **foreign function library** for Python. It provides **C** compatible **data types**, and allows **calling functions** in DLLs or shared libraries. It can be used to **wrap** calls to these libraries in pure Python.

- Python manual: https://docs.python.org/2/library/ctypes.html
- SciPy Lecture: https://scipy-lectures.org/advanced/interfacing_with_c/interfacing_with_c.html#id3
- SciPy Cookbook CTypes: http://scipy.github.io/old-wiki/pages/Cookbook/Ctypes

Caveats:

- Have to compile your code into a shared (dynamic) library;
- Not suitable for complex data types.
- No explicit support for C++ (unimportant !)

CTypes

- **C types**: `c_int, c_double, c_float, c_bool, c_char, c_size_t, ...;`
- **Arrays types**: `(c_int * 10), (c_double * 20);`
- **Pointer types**: `POINTER(c_int), POINTER(c_double);`
- Special **pointer types**: `c_char_p, c_void_p`.
- **Constructors**: `c_int(), c_int(variable), (c_int * 10)()`
- **Pointer** to a variable: `pointer(variable);`
- **Type casting**: `cast(array, POINTER(c_int));`
- **Functions**: `sizeof(variable), sizeof(c_int), addressof(variable);`

CTypes example

```
>>> import ctypes as ct
>>> dir(ct)
…
>>> ct.c_double
>>> ct.c_int
>>> ct.c_char

>>> ct.c_double()

>>> cx = ct.c_double(3.14)
>>> cx

>>> ct.sizeof(cx)

>>> ct.addressof(cx)
>>> hex(ct.addressof(cx))

>>> cx.value
>>> cx.value = 2.72
>>> cx
```

- Load the **module**.
- Check the **contents**.
- CTypes data **types**.
- Create variables using **constructors**.
- Get **information** about CTypes **objects**.
- CTypes **objects** are **mutable**.

CTypes example: Calling an external function.

```
>>> import ctypes as ct
>>> import ctypes.util
>>> dir(ct.util)
...
>>> ct.util.find_library("m")
'libm.so.6'

>>> libm = ctypes.cdll.LoadLibrary("libm.so.6")
>>> dir(libm)
...
>>> libm.cos
<_FuncPtr object at 0x7fb55cb90bb0>

>>> dir(libm)
...

>>> libm.cos.restype = ctypes.c_double
>>> libm.cos.argtypes = [ctypes.c_double]
>>> libm.cos(3.14)
-0.9999987317275395
```

- `util.find_library()` searches **standard locations** for “`libm.so`”.
- **Full path** to the library.
- **Library object** with **lazy access**.
- Must be defined:
  - `argtypes` is a **list of types**.
  - `restype` is a **type**.

C-pointer concept refresher

- **Pointer** is a **variable** containing a generalized **memory address**.
- **Dereference** operator * and **Address** operator &.
- **C-arrays** and **pointers** are very **similar**.
- **Arrays** are **constants** and **pointers** are **variables**.
- Pointers have **types** because they point to data of **specific size** and **format**.

```
double x = 3.14;                  // Variable
double *xp = &x;                  // Pointer to variable

int i = 123;
int *ip = &i;

double xx[] = {1.1, 2.2, 3.3, 4.4, 5.5};  // Array
double *xxp = xx;                         // Pointer

// Accessing data:
printf("%g  %g  %g  %g\n",
       xx[3], xxp[3], *(xx + 3), *(xxp + 3));

char s1[] = "Array string.";
char *s2 = "Pointer string.";

// Compilation error.
s1 = s2;
// Works, but we lose access to the s2 string
s2 = s1;
```

CTypes pointers

```
>>> xx = (ctypes.c_int * 10)()
>>> xx
<__main__.c_int_Array_10 object at 0x7f258551f710>

>>> list(xx)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

>>> xx[:] = range(10)
>>> xx[:]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> px = ctypes.POINTER(ctypes.c_int)(xx)
>>> px
<__main__.LP_c_int object at 0x7f258551f4d0>

>>> px[2]
2
>>> xx[10]
… Index error …
>>> px[10]
33  ← This is garbage, but it works anyways.
```

- **Arrays** and **pointers** for accessing **multiple** data.
- Arrays can be **cast** into pointers **automatically**.
- Arrays know the **limits**, pointers do not.
- There are **many ways** to create a pointer.
- `ctypes.POINTER` is a **constructor**.
- `ctypes.pointer` is a **function**.

Our Case Task

A test case **python code** that

- **Generates** a random configuration of Ar atoms inside a cubic simulation box.
- **Checks** the configuration for spatial overlaps between atoms.
- **Reports** the total number of overlaps in the system.
- The program **accepts** required parameters from the **command line**.

Python code outline: Main logic

```
# === Classes ================================
     ...

# === Functions ==============================
     ...

# === Main code ==============================
# Read input parameters.
params = get_input()
     ...
# Generate a configuration of a Number of Ag atoms.
conf = gen_config(params)
     ...
# Check the configuration for atomic clashes (overlaps).
overlaps = check_overlaps(conf)
     ...
# Report info on found overlaps here.
     ...
timings.report()
# === End of code ============================
```

- 3 main **sections**: **classes**, **functions**, **main** code.
- The main code does **5 things**:
  - gets input **parameters**.
  - generates **random atoms** using the parameters.
  - finds **overlaps** between atoms.
  - **reports** the found overlaps.
  - reports **timings**.
- **Timings** are our data of interest.

Python code outline (cont.): Functions and Classes

```
# === Classes ===============================
class timing_type:
     ...
class params_type:
     ...
class atom_type:
     ...
class config_type:
     ...
# === Functions =============================
def get_input():
     ...
     return params
def gen_config(params):
     ...
     return conf
def check_overlaps(conf):
     ...
     return overlaps
# === Main code =============================
```

- **Storage classes** with some **reporting** capabilities.
- There is a **function** for each major **step**.
- Functions return a **storage object** for the **next step**.
- **Final timings** are reported by the **timing object**.

Python code: How to use.

- Input: **box size** (Å), **number of atoms** (***N***), **random seed**.

  ```
  $ ./overlaps.py 50 1000 0
  ```

- Configuration of **Ar atoms** of **3.4** Å in diameter (***d***).

- Internals of the **overlap detecting** function:
  - Double loop ***i, j*** over the all atoms.
  - Use $(d^2 < r^2)$ for clash condition.
  - Store overlaps as a list of (i, j, d_{ij}) tuples.
  - Use the overcounting to check for correctness, $(N + 2 * N_{OL})$.

- **Prints** out the **first 7** overlaps.

- **Test**, if it works properly.
  - (50, 1000, 0 → **591** overlaps).

What step is the slowest?

- What step is the **slowest**?
  - 10 atoms in a 100 Ang box?
  - 10000 atoms in a 100 Ang box?
- **How slow** is it?
  - Very?
  - A little?
- **How long** will it take for 1 000 000 atoms?
- **What** can we do about it?
  - Rewrite the whole thing in Fortran-77.
  - Find a better computer.

How slow is the slow? Big O notation to describe complexity.

- **O(1)** describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.
  get_input(...) is a **O(1)** complexity function.

- **O(N)** describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set.
  gen_config(...) is a **O(N)** complexity function.

- **O(N^2)** represents an algorithm whose performance is directly proportional to the square of the size of the input data set.
  check_overlaps(...) is an **O(N^2)** complexity function.

- **Total complexity** is the worst complexity of the steps, **O(N^2)** here.

C++ design decisions

Requirements:

- We have to pass **coordinates**, atomic **radii**;
- We obtain **(i, j)** pairs as well as **distances**, and **number** of clashes;
- We do not know the **number** of clashes upfront.

Design:

- Use **basic** C / C++ data types: int, double, array;
- C **arrays** are not aware of their length. We have to pass the **lengths**.
- **Array arguments** are represented by **pointers**;
- Cannot easily “grow” array sizes. Memory should be **allocated**.

C++ design

- **Preallocate** return buffers on the **Python side**;
- **Find all** atomic overlaps;
- Return the **number** of overlaps;
- Return as many (i, j, distance) values as **possible**.
- If the **number** is larger than the **size** of the buffer, **reallocate** the buffers and **redo** the search.

```
int check_overlaps(const double* xx,
                   const double* yy,
                   const double* zz,
                   const double* rr,
                   int n,
                   int maxnolaps,
                   int* ii,
                   int* jj,
                   double* dd) {
    …
    return nolaps;
}
```

Calling C++ code: The Plan

- Place our C++ function to `my.cpp` source file.
- Compile the `my.cpp` source code to `my.o` object file.
- Package the `my.o` object file to `libmycpp.so` shared library.
- Place the ***check_overlaps(...)*** wrapper function into `mycpplib.py` file.
- Load `mycpplib.py` in the main `overlap.py` code as a module.
- Call the wrapper function as ***mycpplib.check_overlaps(...)***.

Calling C++ code: `my.cpp`

- Include `<iostream>`, `<cmath>`.
- Use namespace `std`.
- Create the '`extern "C"`' code block to prevent function **name mangling**.
- Write a dummy ***overlaps(...)*** function.

- Compile to check for errors.
  - `g++ -Wall -fPIC -c my.cpp`

`-Wall` enables lots of warnings on strange code.
`-fPIC` generate position-independent code.

Calling C++ code: Implement overlap search

- ***i, j*** double loop, ***i = [0, n)***, ***j = [i+1, n)***.
- Compare $\boldsymbol{rc^2}$ vs $\boldsymbol{d^2}$, to avoid unnecessary ***sqrt()*** calls.
- Fill up the buffers only until the preallocated mark, ***max_nolaps***.
- **Compile**
  - `g++ -O2 -Wall -fPIC -c my.cpp`
- **Build** a shared library:
  - `g++ -shared -o libmycpp.so my.o`

Calling C++ code: `mycpplib.py`

- Import `sys`, `os`, `math`, `time`, `ctypes`.
- Load the the shared library, `libmycpp.so`.
- Write a dummy ***check_overlaps(...)*** wrapper function.
- Import the `mycpplib.py` module in the main code.
- Change the ***check_overlaps(...)*** call in the main code to the new dummy wrapper function.
- **Test**, that it works properly.

Calling C++ code: Python wrapper function

- Define the **return** and **input** argument **types**.
- Allocate and populate the **input arrays**.
- Allocate **output arrays** using ***max_nolaps*** initial guess.
- Call the C++ function and obtain the true number of overlaps, ***nolaps***.
- Pack and return the list of **overlap** tuples, ***(i, j, d)***.
- Test, that it works properly.
  - (50, 1000, 0 → **591** overlaps) !!! Works.
  - (100, 10000, 0 → **7948** overlaps)
  - (50, 10000, 0 → **20000** overlaps) !!! buffers are too small.

Calling C++ code: Handling number of overlaps greater than max

- Use a loop with post condition: `while True … if … break`;
- If ***nolaps* < *max_nolaps***, then we are done.
- If not, **reallocate** and **recompute**.
- **Double time** in the worst case.
- **Test**, that it works properly.
  - (100, 10000, 0 → **7948** overlaps)
  - (50, 10000, 0 → **60983** overlaps), note the double time.

C++ design. No recompute

- **Allocate** the memory on **C++ side** of the code;
- Have to **free** the **memory** on the **C++ side**.
- Complication:
  - **C arguments** are “**by value**”.
- For return data have to use
  - `POINTER(POINTER(c_int))`
  - `int**`

```
int check_overlaps(const double* xx,
                   const double* yy,
                   const double* zz,
                   const double* rr,
                   int n,
                   int** pii,
                   int** pjj,
                   double** pdd){
    ......................................
    return nolaps;
}

void free_mem(int* ii,
              int* jj,
              double* dd){
    ......................................
}
```

Calling C++ code: Design without recompute

- Include `<vector>`.
- Build from ***overlaps(...)***:
  - copy and rename to ***overlaps_mem(...)***.
  - Change the call parameters: no max and ** pointers.
- Collect the data into an **STL container** that can dynamically **grow**;
- Container is a **local variable** that cannot be returned.
- **Allocate** memory, **copy** the data, update return **pointers**.
- Implement a ***free_mem(...)*** function to free the allocated memory.
- **Compile**
  - `g++ -O2 -Wall -fPIC -c my.cpp`
- **Build** a shared library:
  - `g++ -shared -o libmycpp.so my.o`

Calling C++ code: New Python wrapper

- Using the same source file `mycpplib.py`.
- ***check_overlaps_mem(...)***, based on ***check_overlaps(...)*** wrapper.
- Add `array of pointers` types to the local definitions.
- No ***max_nolaps***.
- Change the call to ***overlaps(...)*** to ***overlaps_mem(...)***.
- Change the return array argument types to **array of pointers**.
- Allocate arrays of pointers, size = 1, ***pii, pjj, pdd***.

Calling C++ code: New Python wrapper (cont.)

- Call the function and obtain the number of overlaps, ***nolaps***.
- Prepare the resulting list of tuples, ***[(i, j, d)]***.
- Use ***pi, pj, pd*** pointers to access data, for convenience.
- **Free memory** using the pointers.
- Change the call from the main code to ***check_overlaps_mem(...)***.
- **Test**, that it works properly.
  - (100, 10000, 0 → **7948** overlaps)
  - (50, 10000, 0 → **60983** overlaps), note the single time.

Conclusions and Observations

- We have been able to significantly **speed up** our test Python code using C++.
- **CTypes** is a relatively easy and **standard way** to call C/C++ functions from Python.
- **Memory management** can be an issue when the size of the return data is unknown.
- Different **design patterns** demonstrate different **performance**.
- Unlike pure Python there is a big difference between available C-functions on **Windows and Linux**. If you are using system libraries you may need to program **different versions** you Python and C++ code for Windows and Linux.

Thank you.

Questions?