Programming and Data Analysis for Scientists

Seminar – Compiled vs Interpreted code

University of BRISTOL

Prof Stephen Clark

SCIF20002

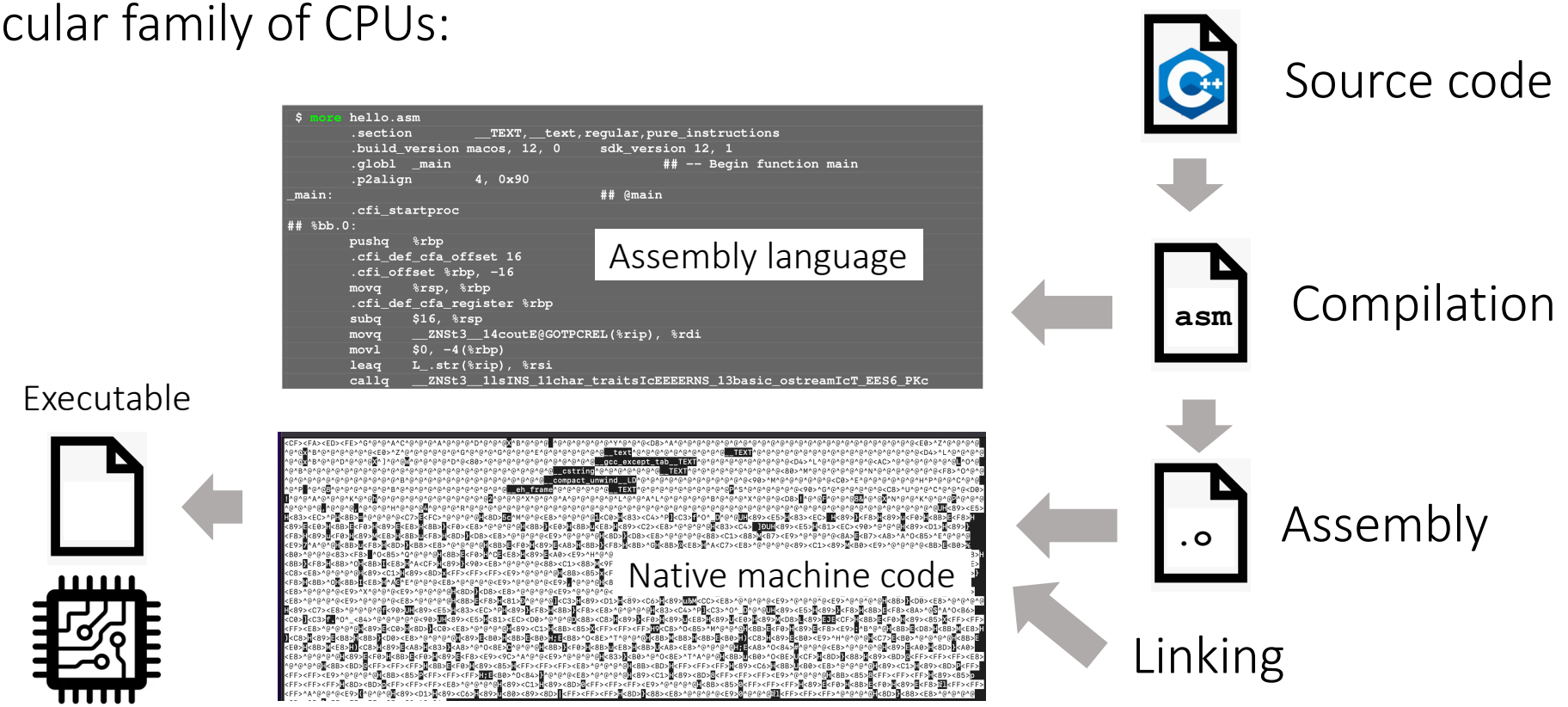
Compiled vs Interpreted code*

- We have been learning C/C++ for a couple of weeks now. A key feature of this language is that code is **compiled** making it **compact** and **fast.**
- Python is much more user-friendly than C/C++ making it extremely easy to prototype and explore new ideas and methodologies (with a big ecosystem of libraries).
- **But**, Python is slow due to being **interpreted**. In particular explicit looping is very slow due to overheads created by Python's dynamic variables type.

***Pedants**: technically whether code is compiled or interpreted is not a property of the *language* itself, but rather of the implementation. You could have a C/C++ interpreter and a Python compiler (although this is difficult to code).

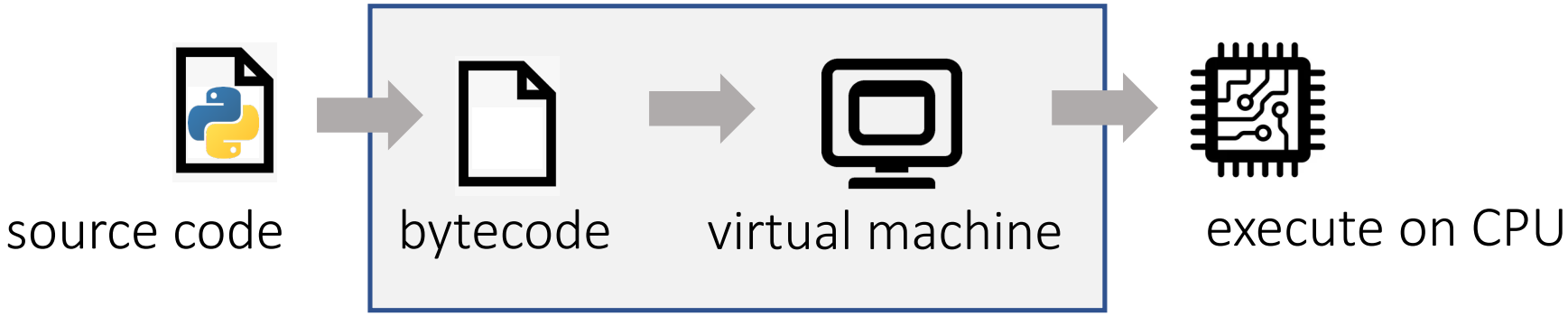
What is compilation?

Compiling takes source code and converts it into binary machine code native to a particular family of CPUs:



What is interpretation?

Interpreted source code is usually “compiled” not into binary machine code but into a bytecode – a kind of machine code for a virtual machine (hardware abstracted). The virtual machine is then a compiled executable that “runs” the bytecode instructions.



This form of virtualization approach was pioneered by *Java* and has become very popular because it is hardware agnostic (“write once run anywhere”).

Simple speed test

Ok, let's do a straightforward speed test between C++ and Python.

Write a Python script (counter.py) and a C++ program (counter.cpp) which iterates a counter from 0 to 1 billion and then prints the value of the counter to the console at the end.

Run and time them on the command line (Mac terminal shown):

```
% time python counter.py
```

```
% g++ counter.cpp -o counter
% time ./counter
```

You should get something like this:

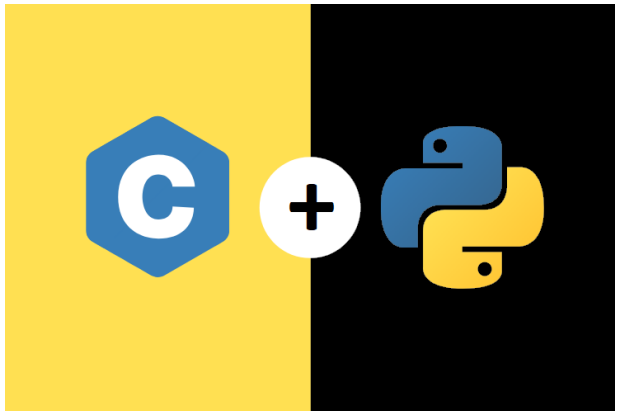
```
1000000000
python counter.py  43.83s user 0.06s system 99% cpu 43.911 total
```

Python is nearly 100 times slower!

```
1000000000
./counter  0.52s user 0.00s system 78% cpu 0.672 total
```

Calling compiled C++ code from Python

If you know which bits of your code are the bottleneck (it is usually clear) then you could write them in C++ (compile it) and then call them from Python.

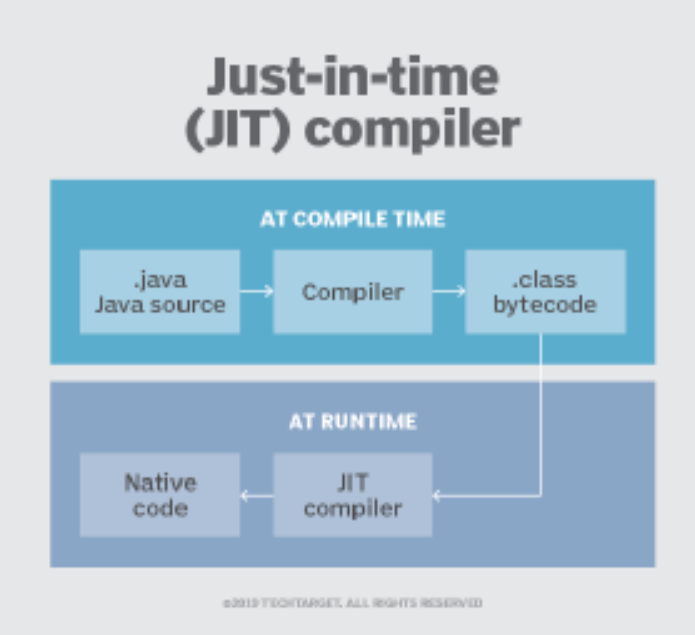


This is a mixed language approach attempting to gain the best of both worlds.

We will have a look at an example using **ctypes** shortly. The issue here is that this kind of binding can be complicated to set up and tricky to debug.

Just in time compilation

An in-between possibility is so called *just in time* compilation. The virtual machine profiles your code to find the “hot-spots” and then compiles these parts during run-time into native machine code.



For numerical calculations the **Numba** package for Python basically does ctypes for you without any effort or recoding.

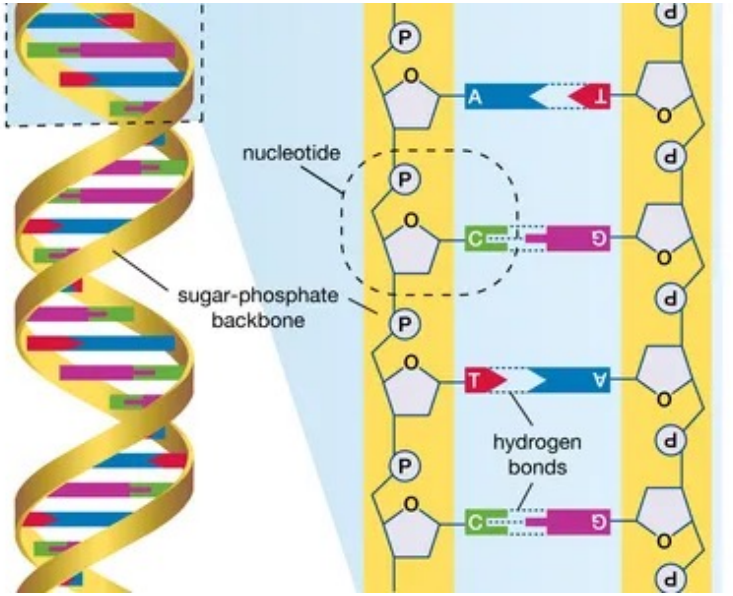
If it works you can get C/C++ similar performance from Python. Let do some more tests …

Example 1: Computing DNA K-mers

A DNA is a long chain of units called nucleotides.

There are 4 types of nucleotides labelled by A, C, G, and T. A short portion DNA could look like:

ACTAGGGATCATGAAGATAATGTTGGTGTTTG



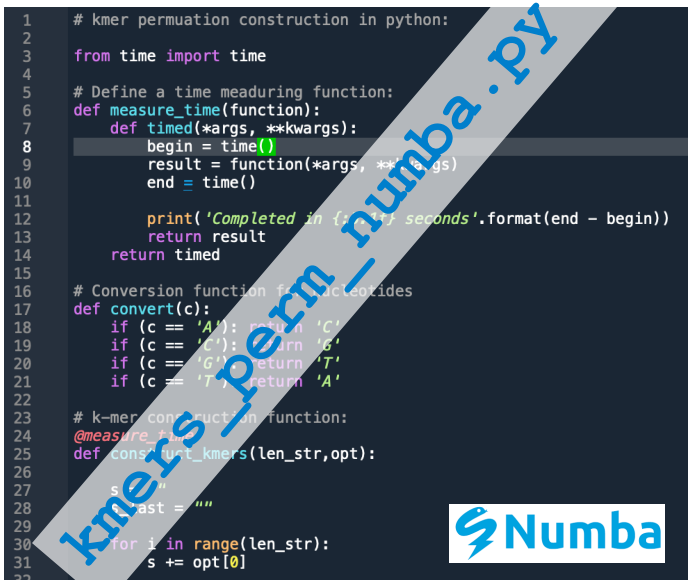
If you choose any K consecutive nucleotides (i.e. letters) from this string, it will be a K-mer. Here are some examples of 4-mers derived from the example.

ACTA, CTAG, TAGG, AGGG, GGGA, etc.

Example 1: The challenge

Let's generate all possible 13-mers. Mathematically it is a permutation with a replacement problem. Therefore, we have 4^{13} (=67,108,864) possible 13-mers.

We will use a simple algorithm to generate results in Python and C++ paying particular attention to how long they take to run (they time themselves) …



=> Run the Python code

=> Compile and run the C++ code

```
g++ --std=c++11 kmers_perm.cpp
-o kmers_perm
```

```
./kmers_perm
```

How do they compare?

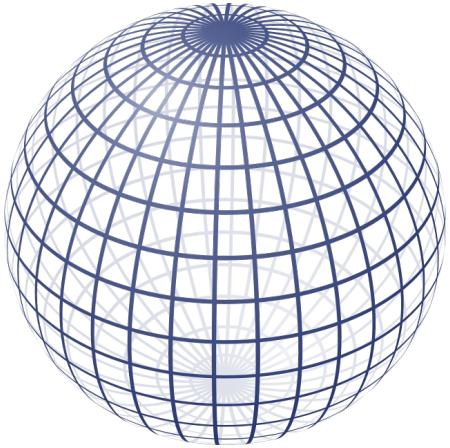
Example 2: Numerical integration in 3D

Let's try to compute the following 3D integral:

3D wave-vector

$$\mathbf{k} = \begin{pmatrix} k_x \\ k_y \\ k_z \end{pmatrix}$$

Constant



$$\int_{\Omega} \frac{\mathrm{d}\mathbf{k}}{2\pi^2} \frac{e^{-\sigma^2 \mathbf{k}^{\mathrm{T}} \epsilon \mathbf{k}}}{\mathbf{k}^{\mathrm{T}} \epsilon \mathbf{k}}$$

Dielectric permittivity tensor

$$\epsilon = \begin{pmatrix} \epsilon_{xx} & 0 & 0 \\ 0 & \epsilon_{yy} & 0 \\ 0 & 0 & \epsilon_{zz} \end{pmatrix}$$

Integrate over a sphere of radius $G_{\max}$

Example 2: Numerical integration in 3D

There are clever methods we could use to do this integration, but let's just do brute force discretisation in Cartesian coordinates …

Use *N* points along each axis so the code will involve:

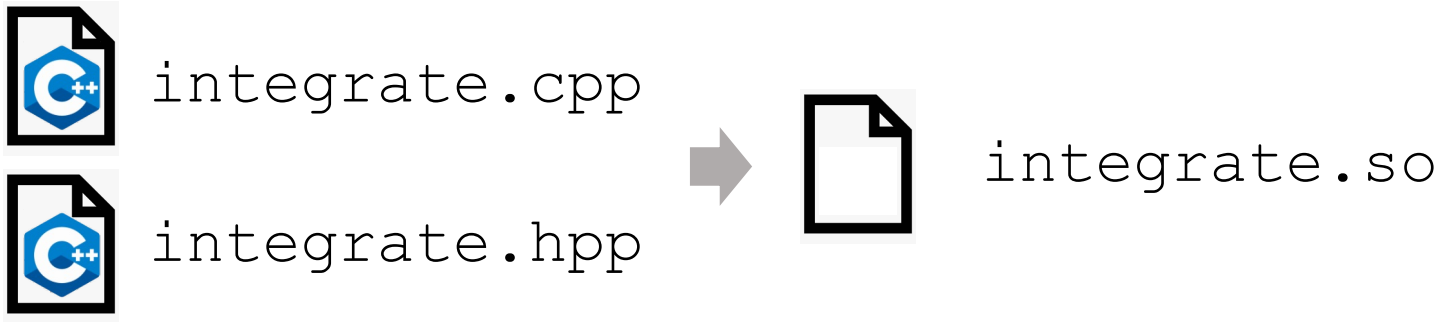
```
loop over k_x
    loop over k_y
        loop over k_z
            if inside sphere then compute
```

Effort to perform nested loops scales as N^3 … will become very slow!

Example 2: Numerical integration in 3D

The code for this example illustrates 3 approaches:

- Fully Python implementation
- Python using **ctypes** binding to a compiled C++ shared library
- C++ code compiling against the C++ shared library



Compile the shared library first as: `g++ -shared integrate.cpp -o integrate.so`

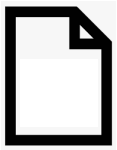
Example 2: Numerical integration in 3D

Run the python script

`compute_integral.py`

Numba

`integrate.hpp`



`integrate.so`

and compile the standalone app:

```
g++ --std=c++11 integrate.so compute_integral.cpp -o compute_integral
```

Will need to specify the dynamic link loader path:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:./
```

```
./compute_integral
```

`compute_integral.cpp`

`compute_integral`

Example 2: Numerical integration in 3D

How do the approaches compare?

We have 4 configurations to compare:

plain Python	> 1000 s got bored waiting!
Python with Numba	65.7 s
Python using integrate.so via ctypes	8.6 s
plain C++	8.5 s

To get C++ performance ultimately needs C++ compiled code to be used.