13 Compiled and interpreted languages

September 13, 2020

1 Introduction

This notebook is optional and intended for those with an interest in going beyond the course material.

Programming languages and commonly classed as *compiled* or *interpreted*. We summarise and demomstrate some of the differences in thie notebook.

1.1 Compiled languages

A compiled language uses a *compiler* to transform input code into a program (machine code) that is executed by a computer. Machine code is the set of instructions for a computer to execute in the CPUs computers 'native' language (instruction set). It is not human readable. The compiler generally processes the entire program, transforming it in a sequence of steps into machine code.

Common compiled languages include C, C++ and Rust.

1.2 Interpreted languages

An interpreted language processes program instructions as they are encountered (line-by-line) rather processing the entire program into machine code ahead of time.

Python in an interpreted language.

1.3 Differences

Compiled languages lead to programs are generally faster than interpretted programs, although in many cases implementations in interpreted language are nowdays fast enough. Compiled programs can have a smaller footprint, which can be important for embedded devices and other platforms with limited capacity. The computer on which a compiled program runs does not need to have a compiler or an interpreter installed.

When a compiler translates code into an executable program it will typiclly perform checks and perform optimisations (static analysis). The compiler checks for valid syntax, and sophiscataed optimisations can perform code transformation to make programs faster. Interpreted languages are usually simpler to develop, and more interactive and avoid the need for a compilation step. Interpreted languages are often dynamically typed, with the interpreter inferring the types, e.g. integers versus floats. With compiled languages types are usually fixed at compile time.

1.4 Just-in-time compilation

The difference between interpreted and compiled languages is not as clear as it once was, with interpreted languages now often using 'just-in-time' compilation. We will explore the impact of compiled code using Numba, a just-in-time compiler for Python. For specific functions that we mark, Numba can compile the code and apply peformance optimisations typical of compiled languages with the objective of making functions faster.

1.5 Objectives

- Understand the difference between compiled and interpreted implementations
- Awareness of intermediate representations and assembly code
- Explore performance differences between interpreted and compiled implementations

We will later use Numba, so we install it now.

[10]:
```
!pip install numba
```

```
Requirement already satisfied: numba in /Users/garth/local/venv-
jupyter/lib/python3.8/site-packages (0.51.2)
Requirement already satisfied: llvmlite<0.35,>=0.34.0.dev0 in
/Users/garth/local/venv-jupyter/lib/python3.8/site-packages (from numba)
(0.34.0)
Requirement already satisfied: numpy>=1.15 in /Users/garth/local/venv-
jupyter/lib/python3.8/site-packages (from numba) (1.19.1)
Requirement already satisfied: setuptools in /Users/garth/local/venv-
jupyter/lib/python3.8/site-packages (from numba) (50.3.0)
```

2 Performance of interpreted and compiled functions

In 07 Numerical computation we tested the performance of a simple function for computing the norm of a long vector. We consider a similar problem here: computing the dot product of a vector with itself, $\boldsymbol{x} \cdot \boldsymbol{x}$, using our own Python function and using NumPy:

[2]:
```
import numpy as np
import random

def compute_norm2(x):
    norm2 = 0.0
    for xi in x:
        norm2 += xi*xi
    return norm2

x = np.random.rand(10000000)
%time n0 = compute_norm2(x)
%time n1 = np.dot(x, x)
```

```
CPU times: user 2.9 s, sys: 9.53 ms, total: 2.91 s
Wall time: 2.92 s
```

```
CPU times: user 24.5 ms, sys: 929 µs, total: 25.5 ms
Wall time: 5.98 ms
```

As expected, the NumPy code is many orders of magnitude faster. NumPy in fact uses compiled code for the computation, which is the reason why it is much faster than our pure Python implementation.

We now make a small change and add the 'decorator' `@numba.jit` to our function. This instructs Numba to transform our function in a compiled function/program.

[15]:
```
import numba

@numba.jit(nopython=True)
def compute_norm2(x):
    norm2 = 0.0
    for xi in x:
        norm2 += xi*xi
    return norm2

x = np.random.rand(10000000)
compute_norm2(x)
%time n0 = compute_norm2(x)
%time n1 = np.dot(x, x)
```

```
CPU times: user 14.6 ms, sys: 33 µs, total: 14.6 ms
Wall time: 14.6 ms
CPU times: user 7.31 ms, sys: 313 µs, total: 7.62 ms
Wall time: 1.85 ms
```

Note that we call `compute_norm2` twice and time only the second call. We want to measure the raw cost of the computation and not the small Numba just-in-time compilation overhead that is incurred the first time a functon is processed.

The Numba version is much faster than the pure Python version. NumPy is faster again for this operation, but relative close to the Numba time. This is likely because NumPy is using a highly optimised BLAS (Basic Linear Algebra Subprograms) implementation, which is a set of machine code level functions that are tuned for numerical computations.

3 Sorting implementations

We saw in 10 Algorithms that our implementation of the quicksort algorithm was considerably slower than the Python built-in quicksort. Part of the performance difference could be explained by our implementation being in pure Python, with the built-in Python function being implemented in a compiled language.

We can explore the difference compilation might make to our implementation. To start, we reproduce the pure Python quicksort implementation:

[4]:
```
def partition_ref(A, lo, hi):
    "Partitioning function for use in quicksort"
```

```python
    pivot = A[hi]
    i = lo
    for j in range(lo,  hi):
        if A[j] <= pivot:
            A[i], A[j] = A[j], A[i]
            i += 1
    A[i], A[hi] = A[hi], A[i]
    return i

def quicksort_ref(A, lo=0, hi=None):
    "Sort A and return sorted array"

    # Initialise data the first time function is called
    if hi is None:
        A = A.copy()
        hi = len(A) - 1

    # Sort
    if lo < hi:
        p = partition_ref(A, lo,  hi)
        quicksort_ref(A, lo, p - 1)
        quicksort_ref(A, p + 1, hi)
    return A
```

We now introduce a version annotated with a Numba decorator:

[5]:
```python
@numba.jit(nopython=True)
def partition_jit(A, lo, hi):
    "Partitioning function for use in quicksort"
    pivot = A[hi]
    i = lo
    for j in range(lo,  hi):
        if A[j] <= pivot:
            A[i], A[j] = A[j], A[i]
            i += 1
    A[i], A[hi] = A[hi], A[i]
    return i

@numba.jit(nopython=True)
def quicksort_jit(A, lo=0, hi=-1):
    "Sort A and return sorted array"

    # Initialise data the first time function is called
    if hi == -1:
        A = A.copy()
        hi = len(A) - 1
```

```python
    # Sort
    if lo < hi:
        p = partition_jit(A, lo,  hi)
        quicksort_jit(A, lo, p - 1)
        quicksort_jit(A, p + 1, hi)
    return A
```

The last argument to `quicksort_jit` has been changed slightly so that the argument type does not change (argument types that change are problematic for a compiler as it needs to know ahead of time which types to generate machine code for).

We can now time our pure Python implementation, the Numba-compiled implementation and the built-in sort function. As before, we will call `quicksort_jit` once before timing to eliminate the cost of the just-in-time compilation.

[17]:
```python
data = np.random.rand(500000)

# Time the pure Python implementation
%time x = quicksort_ref(data)

# Time the Numba implementation
quicksort_jit(data)
%time x = quicksort_jit(data)

# Time the built-in implementation
%time x = np.sort(data, kind='quicksort')
```

```
CPU times: user 5.24 s, sys: 14.3 ms, total: 5.26 s
Wall time: 5.28 s
CPU times: user 58.4 ms, sys: 332 µs, total: 58.8 ms
Wall time: 58.9 ms
CPU times: user 33.8 ms, sys: 221 µs, total: 34.1 ms
Wall time: 33.9 ms
```

The pure Python implementation is clearly the slowest. The Numba and built-in implementation are relatively closde in time. Note that the Numba implementation is virtually a direct translation of the pure Python implementation and has not been carefully optimised.

4 Intermediate representations and assembly code

A compiler translates input code into (i) an 'intermediate representation' (IR), and then into (ii) machine code. The IR is the compiler's internal representation of a program. A compiler can perform optimisations on the IR that may make a program faster and which may be specific to the CPU type. Machine code is the low instructions sent to the CPU.

With Numba we can inspect the IR and the assembly code. Assembly code is human readable code (but very low level) that maps almost one-to-one to machine code (which would be very hard to read).

Consider a very simple function that returns the sum of two integers:

[7]:
```
from numba import int64

@numba.jit('int64(int64, int64)', nopython=True)
def add(x, y):
    return x + y

add(2, 3)
```

[7]: 5

Not that we have specified the argument types in this case.

We can inspect the compiler's IR for the this function:

[18]:
```
for v, k in add.inspect_llvm().items():
    print(k)
```

```
; ModuleID = 'add'
source_filename = "<string>"
target datalayout =
"e-m:o-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-darwin19.6.0"

@"_ZN08NumbaEnv8__main__7add$247Exx" = common local_unnamed_addr global i8* null
@.const.add = internal constant [4 x i8] c"add\00"
@PyExc_RuntimeError = external global i8
@".const.missing Environment: _ZN08NumbaEnv8__main__7add$247Exx" = internal
constant [55 x i8] c"missing Environment: _ZN08NumbaEnv8__main__7add$247Exx\00"

; Function Attrs: nofree norecurse nounwind writeonly
define i32 @"_ZN8__main__7add$247Exx"(i64* noalias nocapture %retptr, { i8*,
i32, i8* }** noalias nocapture readnone %excinfo, i64 %arg.x, i64 %arg.y)
local_unnamed_addr #0 {
entry:
  %.14 = add nsw i64 %arg.y, %arg.x
  store i64 %.14, i64* %retptr, align 8
  ret i32 0
}

define i8* @"_ZN7cpython8__main__7add$247Exx"(i8* nocapture readnone
%py_closure, i8* %py_args, i8* nocapture readnone %py_kws) local_unnamed_addr {
entry:
  %.5 = alloca i8*, align 8
  %.6 = alloca i8*, align 8
  %.7 = call i32 (i8*, i8*, i64, i64, …) @PyArg_UnpackTuple(i8* %py_args, i8*
getelementptr inbounds ([4 x i8], [4 x i8]* @.const.add, i64 0, i64 0), i64 2,
i64 2, i8** nonnull %.5, i8** nonnull %.6)
```

```
  %.8 = icmp eq i32 %.7, 0
  br i1 %.8, label %entry.if, label %entry.endif, !prof !0

entry.if:                                         ; preds =
%entry.endif.endif.endif.endif.endif, %entry.endif.endif.endif, %entry
  ret i8* null

entry.endif:                                      ; preds = %entry
  %.12 = load i8*, i8** @"_ZN08NumbaEnv8__main__7add$247Exx", align 8
  %.17 = icmp eq i8* %.12, null
  br i1 %.17, label %entry.endif.if, label %entry.endif.endif, !prof !0

entry.endif.if:                                   ; preds = %entry.endif
  call void @PyErr_SetString(i8* nonnull @PyExc_RuntimeError, i8* getelementptr
inbounds ([55 x i8], [55 x i8]* @".const.missing Environment:
_ZN08NumbaEnv8__main__7add$247Exx", i64 0, i64 0))
  ret i8* null

entry.endif.endif:                                ; preds = %entry.endif
  %.21 = load i8*, i8** %.5, align 8
  %.24 = call i8* @PyNumber_Long(i8* %.21)
  %.25 = icmp eq i8* %.24, null
  br i1 %.25, label %entry.endif.endif.endif, label %entry.endif.endif.if, !prof
!0

entry.endif.endif.if:                             ; preds = %entry.endif.endif
  %.27 = call i64 @PyLong_AsLongLong(i8* nonnull %.24)
  call void @Py_DecRef(i8* nonnull %.24)
  br label %entry.endif.endif.endif

entry.endif.endif.endif:                          ; preds = %entry.endif.endif,
%entry.endif.endif.if
  %.22.0 = phi i64 [ %.27, %entry.endif.endif.if ], [ 0, %entry.endif.endif ]
  %.32 = call i8* @PyErr_Occurred()
  %.33 = icmp eq i8* %.32, null
  br i1 %.33, label %entry.endif.endif.endif.endif, label %entry.if, !prof !1

entry.endif.endif.endif.endif:                    ; preds =
%entry.endif.endif.endif
  %.37 = load i8*, i8** %.6, align 8
  %.40 = call i8* @PyNumber_Long(i8* %.37)
  %.41 = icmp eq i8* %.40, null
  br i1 %.41, label %entry.endif.endif.endif.endif.endif, label
%entry.endif.endif.endif.endif.if, !prof !0

entry.endif.endif.endif.endif.if:                 ; preds =
%entry.endif.endif.endif.endif
  %.43 = call i64 @PyLong_AsLongLong(i8* nonnull %.40)
```

```
  call void @Py_DecRef(i8* nonnull %.40)
  br label %entry.endif.endif.endif.endif.endif

entry.endif.endif.endif.endif.endif:              ; preds =
%entry.endif.endif.endif.endif, %entry.endif.endif.endif.endif.if
  %.38.0 = phi i64 [ %.43, %entry.endif.endif.endif.endif.if ], [ 0,
%entry.endif.endif.endif.endif ]
  %.48 = call i8* @PyErr_Occurred()
  %.49 = icmp eq i8* %.48, null
  br i1 %.49, label %entry.endif.endif.endif.endif.endif.endif, label %entry.if,
!prof !1

entry.endif.endif.endif.endif.endif.endif:        ; preds =
%entry.endif.endif.endif.endif.endif
  %.14.i = add nsw i64 %.38.0, %.22.0
  %.74 = call i8* @PyLong_FromLongLong(i64 %.14.i)
  ret i8* %.74
}

declare i32 @PyArg_UnpackTuple(i8*, i8*, i64, i64, …) local_unnamed_addr

declare void @PyErr_SetString(i8*, i8*) local_unnamed_addr

declare i8* @PyNumber_Long(i8*) local_unnamed_addr

declare i64 @PyLong_AsLongLong(i8*) local_unnamed_addr

declare void @Py_DecRef(i8*) local_unnamed_addr

declare i8* @PyErr_Occurred() local_unnamed_addr

declare i8* @PyLong_FromLongLong(i64) local_unnamed_addr

; Function Attrs: norecurse nounwind readnone
define i64 @"cfunc._ZN8__main__7add$247Exx"(i64 %.1, i64 %.2) local_unnamed_addr
#1 {
entry:
  %.14.i = add nsw i64 %.2, %.1
  ret i64 %.14.i
}

; Function Attrs: nounwind
declare void @llvm.stackprotector(i8*, i8**) #2

attributes #0 = { nofree norecurse nounwind writeonly }
attributes #1 = { norecurse nounwind readnone }
attributes #2 = { nounwind }
```

```
!0 = !{!"branch_weights", i32 1, i32 99}
!1 = !{!"branch_weights", i32 99, i32 1}
```

The IR would be of interest to someone designing compilers or seeing the optimisation transformations that a compiler might perform.

In some very special cases it can be helpful to inspect the assembly code, which is the closest to readable version of CPU instructions. It is usually inspected only in cases where an understanding of the lowest level operations is required, e.g. when extreme performance is necessary. It is specific to a CPU architecture.

[9]:
```python
for v, k in add.inspect_asm().items():
    print(k)
```

```
        .section        __TEXT,__text,regular,pure_instructions
        .macosx_version_min 10, 15
        .globl  __ZN8__main__7add$247Exx
        .p2align        4, 0x90
__ZN8__main__7add$247Exx:
        addq    %rcx, %rdx
        movq    %rdx, (%rdi)
        xorl    %eax, %eax
        retq

        .globl  __ZN7cpython8__main__7add$247Exx
        .p2align        4, 0x90
__ZN7cpython8__main__7add$247Exx:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        pushq   %r15
        .cfi_def_cfa_offset 24
        pushq   %r14
        .cfi_def_cfa_offset 32
        pushq   %r13
        .cfi_def_cfa_offset 40
        pushq   %r12
        .cfi_def_cfa_offset 48
        pushq   %rbx
        .cfi_def_cfa_offset 56
        subq    $24, %rsp
        .cfi_def_cfa_offset 80
        .cfi_offset %rbx, -56
        .cfi_offset %r12, -48
        .cfi_offset %r13, -40
        .cfi_offset %r14, -32
        .cfi_offset %r15, -24
        .cfi_offset %rbp, -16
```

```
        movq    %rsi, %rdi
        movabsq $_.const.add, %rsi
        movabsq $_PyArg_UnpackTuple, %rbp
        leaq    16(%rsp), %r8
        leaq    8(%rsp), %r9
        movl    $2, %edx
        movl    $2, %ecx
        xorl    %eax, %eax
        callq   *%rbp
        testl   %eax, %eax
        je      LBB1_1
        movabsq $__ZN08NumbaEnv8__main__7add$247Exx, %rax
        cmpq    $0, (%rax)
        je      LBB1_4
        movq    16(%rsp), %rdi
        movabsq $_PyNumber_Long, %r13
        callq   *%r13
        movabsq $_PyLong_AsLongLong, %r15
        movabsq $_Py_DecRef, %r12
        testq   %rax, %rax
        je      LBB1_6
        movq    %rax, %rbx
        movq    %rax, %rdi
        callq   *%r15
        movq    %rax, %r14
        movq    %rbx, %rdi
        callq   *%r12
        movabsq $_PyErr_Occurred, %rbp
        callq   *%rbp
        testq   %rax, %rax
        jne     LBB1_1
LBB1_9:
        movq    8(%rsp), %rdi
        callq   *%r13
        testq   %rax, %rax
        je      LBB1_10
        movq    %rax, %rbx
        movq    %rax, %rdi
        callq   *%r15
        movq    %rax, %r15
        movq    %rbx, %rdi
        callq   *%r12
        callq   *%rbp
        testq   %rax, %rax
        jne     LBB1_1
LBB1_13:
        addq    %r14, %r15
        movabsq $_PyLong_FromLongLong, %rax
```

```
        movq    %r15, %rdi
        callq   *%rax
LBB1_2:
        addq    $24, %rsp
        popq    %rbx
        popq    %r12
        popq    %r13
        popq    %r14
        popq    %r15
        popq    %rbp
        retq
LBB1_4:
        movabsq $_PyExc_RuntimeError, %rdi
        movabsq $"_.const.missing Environment:
_ZN08NumbaEnv8__main__7add$247Exx", %rsi
        movabsq $_PyErr_SetString, %rax
        callq   *%rax
LBB1_1:
        xorl    %eax, %eax
        jmp     LBB1_2
LBB1_6:
        xorl    %r14d, %r14d
        movabsq $_PyErr_Occurred, %rbp
        callq   *%rbp
        testq   %rax, %rax
        je      LBB1_9
        jmp     LBB1_1
LBB1_10:
        xorl    %r15d, %r15d
        callq   *%rbp
        testq   %rax, %rax
        je      LBB1_13
        jmp     LBB1_1
        .cfi_endproc

        .globl  _cfunc._ZN8__main__7add$247Exx
        .p2align        4, 0x90
_cfunc._ZN8__main__7add$247Exx:
        leaq    (%rdi,%rsi), %rax
        retq

        .comm   __ZN08NumbaEnv8__main__7add$247Exx,8,3
        .section        __TEXT,__const
_.const.add:
        .asciz  "add"

        .p2align        4
"_.const.missing Environment: _ZN08NumbaEnv8__main__7add$247Exx":
```

```
        .asciz  "missing Environment: _ZN08NumbaEnv8__main__7add$247Exx"

.subsections_via_symbols
```

5 Exercises

Select exercises from the previous notebooks that could be made faster using Numba and investigate what speed-ups you can achieve.