Optimización de código Python
Conceptos básicos

Oscar Peredo

ALGES Lab, AMTC, Universidad de Chile

Septiembre 2014 - Santiago, Chile

Esquema

Introducción

Profiling

Minimizar operaciones de larga latencia

Minimizar saltos y llamados a funciones

Maximizar el trabajo dentro de un loop

Bibliografía

Outline

Introducción

Profiling

Minimizar operaciones de larga latencia

Minimizar saltos y llamados a funciones

Maximizar el trabajo dentro de un loop

Bibliografía

Introducción

- ¿Como podemos acelerar nuestras aplicaciones desarrolladas en Python usando un procesador de propósito general?
- ¿Podemos obtener un buen *speedup* solamente re-programando nuestro código?
- **Optimización de código**: conjunto de técnicas y modificaciones de código (dependientes e independientes del hardware) que nos ayudan a acelerar una aplicación.

Contexto

- Dos tipos de desarrolladores de software: orientados a la funcionalidad y orientados al rendimiento (reducen el tiempo de cómputo manteniendo las funcionalidades).
- A veces hay que compatibilizar ambos mundos (*trade-off*)
- Compiladores e interpretes ya producen código optimizado... para determinado hardware.
- ... sin embargo, los compiladores e interpretes a veces fallan.
- Idea: escribir código *compiler/interpreter-friendly*.

Ejemplos

Minimizar operaciones de larga latencia:

```
def trigon(seed,N,points):
    random.seed(seed)
    for i in range(1,N):
        r=random.random()
        d=0.0
        for j in range(0,points):
            x=r*math.cos(d)
            y=r*math.sin(d)
            if(i==100 and j==100):
                print(x,end="")
                print(y)
            d=d+2.0*math.pi/points
```

Tiempo original : 3.46 [s]
Tiempo optimizado: 1.42 [s]
Speedup : 2.4x

Ejemplos

Minimizar saltos y llamados a funciones:

```
def decrement(value,vec,i,lim,div):
   if(vec[i]>=lim):
      return value
   else:
      return value-vec[i]/div
def increment(value,vec,i,lim,div):
   if(vec[i]<lim):
      return value+vec[i]/div
   else:
      return value
def sumvec(vec,n,lim,div):
   value=0.0
   for i in range(0,n):
      if(i%2==0):
         value=increment(value,vec,i,lim,div)
      else:
         value=decrement(value,vec,i,lim,div)
   return value
```

Tiempo original : 1.18 [s]
Tiempo optimizado: 0.43 [s]
Speedup : 2.8x

Ejemplos

Maximizar el trabajo dentro de un loop:

```
def multmat(X,Y,result):
   for i in range(len(X)):
      for j in range(len(Y[0])):
         for k in range(len(Y)):
            result[i][j] += X[i][k] * Y[k][j]

X = [[12,7,3],[4 ,5,6],[7 ,8,9]]
Y = [[5,8,1,2],[6,7,3,0],[4,5,9,1]]
for it in range(1,100000):
   # result 3x4
   result = [[0,0,0,0],[0,0,0,0],[0,0,0,0]]
   multmat(X,Y,result)
```

Tiempo original : 1.35 [s]
Tiempo optimizado: 0.35 [s]
Speedup : 3.8x

Outline

Introducción

Profiling

Minimizar operaciones de larga latencia

Minimizar saltos y llamados a funciones

Maximizar el trabajo dentro de un loop

Bibliografía

Profiling

- Para optimizar nuestro código, necesitamos identificar los cuellos de botella.
- *Code profiler*: herramienta que nos entrega información fina sobre la ejecución de una aplicación (tiempo de ejecucuión, contadores hardware, energía,...).
- Herramientas disponibles para Python:
  `https://docs.python.org/3.2/library/profile.html`
  `https://github.com/rkern/line_profiler`
  `https://zapier.com/engineering/profiling-python-boss/`
  ...
- Para esta sesión, utilizaremos un *line profiler* (**información de tiempo por cada línea de código**)

```
Timer unit: 1e-06 s
File: trigon_v01.py
Function: trigon at line 5
Total time: 26.1105 s
Line#  Hits    Time Per Hit % Time Line Contents
==============================================
  5                                           @profile
  6                                           def trigon(seed,N,points):
  7       1      17    17.0    0.0  random.seed(seed)
  8    1000     707     0.7    0.0  for i in range(1,N):
  9     999     838     0.8    0.0   r=random.random()
 10     999     707     0.7    0.0   d=0.0
 11 5994999 4304544     0.7   16.5   for j in range(0,points):
 12 5994000 5742250     1.0   22.0    x=r*math.cos(d)
 13 5994000 5683941     0.9   21.8    y=r*math.sin(d)
 14 5994000 4541313     0.8   17.4    if(i==100 and j==100):
 15       1      19    19.0    0.0      print(x,end="")
 16       1      32    32.0    0.0      print(y)
 17 5994000 5836096     1.0   22.4    d=d+2.0*math.pi/points
```

Outline

Introducción

Profiling

Minimizar operaciones de larga latencia

Minimizar saltos y llamados a funciones

Maximizar el trabajo dentro de un loop

Bibliografía

Minimizar operaciones de larga latencia

- Algunas partes del código pueden ser muy intensivas en cómputo (tiempo):
  - Expresiones aritméticas complejas
  - Cómputo duplicado
  - Funciones demasiado generales
  - Funciones del sistema operativo
  - Demasiados llamados a librerias
  - ...
- Podemos re-escribir el código para que ejecute más rápido evitando estas partes.

Memoization

Memoization (wikipedia):

In computing, memoization is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

Ejemplo: `trigon`

Vamos al código...

Outline

Introducción

Profiling

Minimizar operaciones de larga latencia

Minimizar saltos y llamados a funciones

Maximizar el trabajo dentro de un loop

Bibliografía

Minimizar saltos y llamados a funciones

- Las instrucciones de salto controlan el flujo en la ejecución de un programa: if-then-else, for, while, switch, llamados a funciones, métodos y atributos en objetos, ...
- Quiebran el procesamiento secuencial implicito de las instrucciones a nivel *assembler*.
- Tipicamente, representan 10%-20% de las instrucciones en un código de propósito general.
- En los procesadores actuales (segmentados superescalares), representan una pérdida significativa de rendimiento.

Branch prediction

Branch predictor (wikipedia):
In computer architecture, a branch predictor is a digital circuit that tries to guess which way a branch (e.g. an if-then-else structure) will go before this is known for sure. The purpose of the branch predictor is to improve the flow in the instruction pipeline. Branch predictors play a critical role in achieving high effective performance in many modern pipelined microprocessor architectures such as x86.

Inlining

Inlining (wikipedia):
In computing, inline expansion, or inlining, is a manual or compiler optimization that replaces a function call site with the body of the callee. This optimization typically improves time and space usage at runtime, at the cost of increasing the final size of the program (i.e. the binary file size)...

Ejemplo: `sumvec`

Vamos al código...

Outline

Introducción

Profiling

Minimizar operaciones de larga latencia

Minimizar saltos y llamados a funciones

Maximizar el trabajo dentro de un loop

Bibliografía

Maximizar el trabajo dentro de un loop

- En sentencias iterativas siempre hay una pérdida de rendimiento (*overhead*): mantener contadores del loop, chequeo de límite de operaciones, saltos condicionales en el principio del loop, ...
- Si el cuerpo del loop es pequeño, el overhead será relativamente alto.
- Solución: incrementar el trabajo útil en cada iteración y fusionar varias iteraciones en una (el número de iteraciones fusionadas se llama *unroll degree*)
- Para pequeño número de iteraciones, podemos aplicar *full unroll* (el número máximo de iteraciones se debe conocer a priori).

Loop unrolling

Loop unrolling (wikipedia):
Loop transformation technique that attempts to optimize a program's execution speed at the expense of its binary size (space-time tradeoff). The transformation can be undertaken manually by the programmer or by an optimizing compiler.

Ejemplo: `matmult`

Vamos al código...

Outline

Introducción

Profiling

Minimizar operaciones de larga latencia

Minimizar saltos y llamados a funciones

Maximizar el trabajo dentro de un loop

Bibliografía

Bibliography

- *Awareness of Architecture in Programming*, *Computer Architecture* (UPC-Barcelona master courses).
- *Programación Consciente de la Arquitectura*, (UChile-DCC curso electivo, CC5320).
- Bryant, R. and O'Hallaron, D., *Computer Systems: A Programmer's Perspective*, Pearson International 2nd Edition, 2011/2.
- Drepper, U., *What Every Programmer Should Know About Memory*, Red Hat Inc., 2007.
- Intel 64 and IA-32 Architectures Optimization Reference Manual (March 2014).
- Hennessy, J. and Patterson, D., *Computer Architecture: A Quantitative Approach*, Morgan-Kauffman 4th Edition, 2007.

Thanks for your attention!

Contact: operedo [at] alges.cl

http://www.alges.cl