Experiencias con Python y CUDA en Computación de Altas Prestaciones

Sergio Armas, Lionel Mena, Alejandro Samarín, Vicente Blanco*,Alberto Morales y Francisco Alm

July 12, 2011

Abstract

El aprovechamiento de la capacidad de cómputo de los dispositivos gráficos para resolver problemas computacionalmente complejos está en auge. El alto grado de paralelismo que esta arquitectura provee, además de la disponibilidad de kits especializados de desarrollo de software para el público general, abren la puerta a nuevas formas de resolver problemas científicos en una fracción del tiempo que emplearían algoritmos similares basados en CPU. El siguiente paso es encontrar el equilibrio entre la potencia de estos paradigmas de programación y la flexibilidad de los lenguajes modernos. Es aquí donde PyCUDA entra en escena; un "wrapper" de la programación CUDA para Python, de forma que ofrece al programador el acceso a la computación de altas prestaciones sobre dispositivos gráficos sin abandonar la comodidad y el dinamismo de este lenguaje (orientación a objetos, tipado dinámico, intérprete interactivo, etc.). Nuestros objetivos se centran en, por un lado, preparar una máquina de prueba equipada con el hardware necesario y, por otro, comprobar las facilidades que promete PyCUDA así como su rendimiento frente a problemas reales.

1 Introducción

CUDA es una arquitectura blablabla ...utilizando PyCUDA [1]
y una gráfica 1
y un código 1

References

[1] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan C. Catanzaro, Paul Ivanov, and Ahmed Fasih, ``Pycuda: Gpu run-time code generation for high-performance computing,'' *CoRR*, vol. abs/0911.3456, 2009.

*Dpto. Estadística, I.O. y Computación, Univ. La Laguna, e-mail: `vblanco@ull.es`

Figure 1: Diagrama de bloques de una GPU Tesla M2070 (Fermi)

Listing 1: Codigo de filtros en Python

```
#!/opt/python2.7/bin/python

import sys
import Image
from abc import ABCMeta, abstractmethod

#                                       #

class Filter:
    """ Clase padre de filtros """
    __metaclass__ = ABCMeta

    # Constantes para indicar con que dispositivo se debe procesar el filtro
    CPU = 0
    CUDA = 1
    OPENCL = 2

    # Atributos
    images = []
    post_img = None

    def __init__(self, *images):
        for im in images:
            self.images.append(im)

    # TODO Esquema de colores como parametro
    def new_post_img(self, mode, size):
        self.post_img = Image.new(mode, size)

    def fetch_result(self):
        return self.post_img

    @abstractmethod
    def Apply(self):
        pass

#                                       #

class ErosionFilter(Filter):
    def __init__(self, *images):
        super(ErosionFilter, self).__init__(*images)

    def Apply(self):
        pass

#                                       #

class DifferenceFilter(Filter):
    def __init__(self, *images):
        super(DifferenceFilter, self).__init__(*images)

    def Apply(self, mode):
        self.new_post_img(self.images[0].mode,
                          (self.images[0].size[0], self.images[0].size[1]))
        for x in xrange(self.images[0].size[0]):
            for y in xrange(self.images[0].size[1]):
                # "diff" resultara ser una tupla de 3 elementos (en imagenes RGB) con la
                # diferencia en valor absoluto por cada canal en ese pixel, comparado con el
                # mismo pixel de la imagen anterior
                diff = tuple([abs(a   b) for a,b in zip(self.images[0].getpixel((x, y)), self.images[1].getpixel((x, y)))]
                # img.putpixel((x, y), value)
                self.post_img.putpixel((x, y), diff)

#                                       #

im1 = Image.open(sys.argv[1])
im2 = Image.open(sys.argv[2])
print sys.argv[1], ": ", im1.format, im1.size, im1.mode, '\n'
diferencia = DifferenceFilter(im1, im2)
diferencia.Apply(Filter.CPU)
post = diferencia.fetch_result()
post.save("post.png", "PNG")
```