INTRODUCCIÓN AL GPU COMPUTING:

TRABAJO PRÁCTICO FINAL

ALUMNO :MATÍAS CINCUNEGUI

INTRODUCCIÓN

A continuación, se presenta la resolución del trabajo práctico final de la materia. Las especificaciones técnicas de su desarrollo son idénticas a las de mi trabajo práctico de cursada, y serán repetidas a continuación:

La solución fue desarrollada en Linux empleando la biblioteca de OpenCL. El hardware utilizado para las pruebas constó del procesador de una computadora de escritorio y una placa de video ATI Radeon HD 6670 como dispositivo principal. Cuando se habla de optimización respecto al procesador, se hace referencia a la diferencia entre un programa ejecutada en éste respecto a la ejecución del mismo programa en la placa de video.

De acuerdo a lo capturado por OpenCL, las especificaciones de cada dispositivo son:

```
Device 1: Turks [GPU]
    Device Version     : OpenCL 1.2 AMD-APP (1445.5)
    OpenCL C Version   : OpenCL C 1.2
    Compute Units      : 6
    Max Work Group Size: 256
    Clock Frequency    : 800
    Local Memory Size  : 32768
    Global Memory Size : 536870912
    Double Precision   : no

  Device 2: AMD Athlon(tm) II X2 215 Processor [PROCESADOR]
    Device Version     : OpenCL 1.2 AMD-APP (1445.5)
    OpenCL C Version   : OpenCL C 1.2
    Compute Units      : 2
    Max Work Group Size: 1024
    Clock Frequency    : 2700
    Local Memory Size  : 32768
    Global Memory Size : 4145164288
    Double Precision   : yes
```

TRABAJO PRÁCTICO FINAL: PATTERN MATCHING

En el presente trabajo se implementó una versión en GPU de un algoritmo de pattern matching provisto por la cátedra. Este algoritmo permite, a partir de una imagen y otra llamada “template”, obtener la sección de la primera imagen que más semejanza tiene con la imagen “template”. Se utiliza, como en el ejemplo provisto por la cátedra, para encontrar una sub-imagen dentro de una imagen más grande.

La versión del algoritmo provista por la cátedra ejecuta en CPU, y consta básicamente de un bucle que, para cada píxel de la imagen principal (respetando los límites de la imagen), itera a partir de ese píxel en un espacio igual al de la imagen template y calcula las diferencias absolutas entre ambos sectores. El sector con menor diferencia encontrado es el que será señalado como matching. Se trata de un doble bucle (de cuatro sentencias for anidadas), de lo cual se sigue lógicamente que para grandes volúmenes de datos, puede resultar lento en el procesador.

La versión GPU implementada parte del mismo principio, pero aprovecha, obviamente, las prestaciones de la concurrencia.

Esta versión ejecuta el primer par de sentencias for (para cada píxel de la imagen, sin pasarse de los límites), corriendo un **kernel** por cada una de ellas. Dentro de cada kernel, está el doble for que calcula la diferencia y, con la ayuda de una variable auxiliar y la sentencia *atomic_min* de OpenCL, las compara con los resultados de otras instancias del kernel que corren en paralelo.

El algoritmo en sí es idéntico a la versión de CPU.

INSTRUCCIONES DE EJECUCIÓN DEL PROGRAMA

El programa puede ser corrido por consola y acepta un parámetro indicando si se desea correrlo en su versión GPU o CPU (versión provista por la cátedra). Si no se le pasa este parámetro, corre por defecto en GPU. El programa, al final, imprime el tiempo que tomó la ejecución (ya sea versión de CPU o GPU).

```
Ejemplos de uso:
$ ./TPFinal                 //corre en gpu
$ ./TPFinal gpu
$ ./TPFinal cpu             //corre en cpu
```

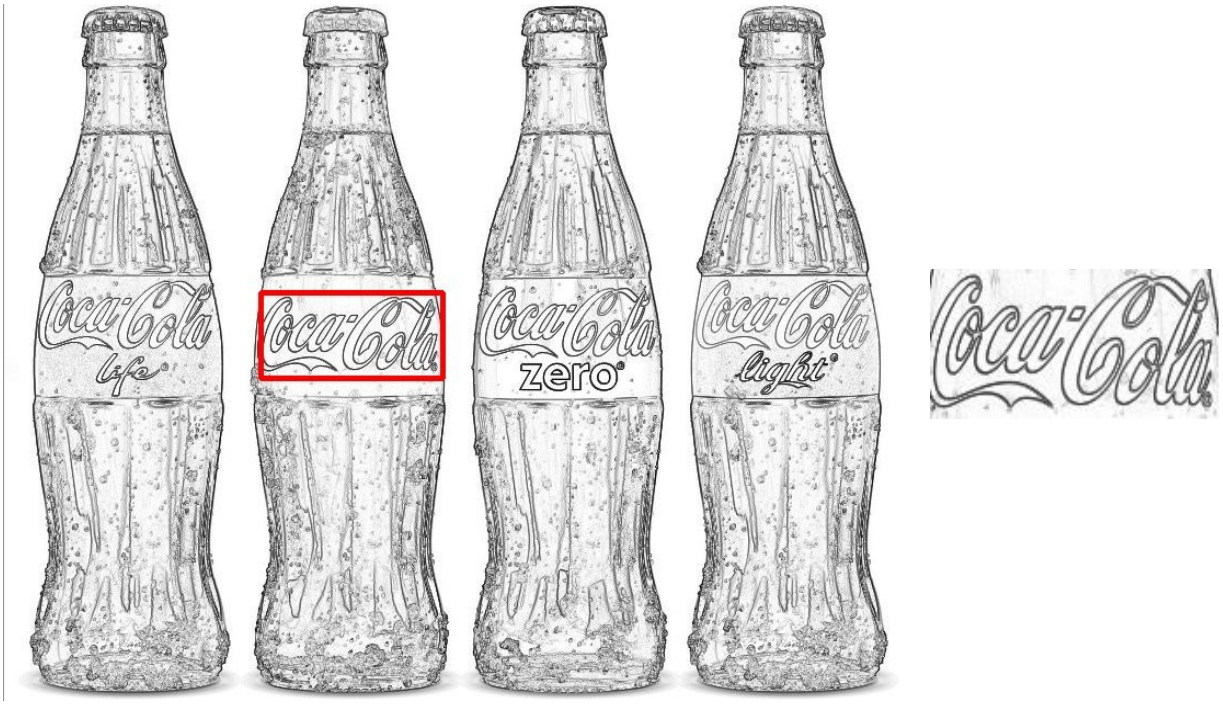
El programa levantará automáticamente las imágenes “baseImage.jpg” y “template.jpg” que se depositen en el directorio donde está el ejecutable. Si se desea probar una imagen distinta, hay que nombrarlas adecuadamente y ponerlas allí. **Ambas imágenes deben estar en formato jpg.**

ANÁLISIS DE RESULTADOS

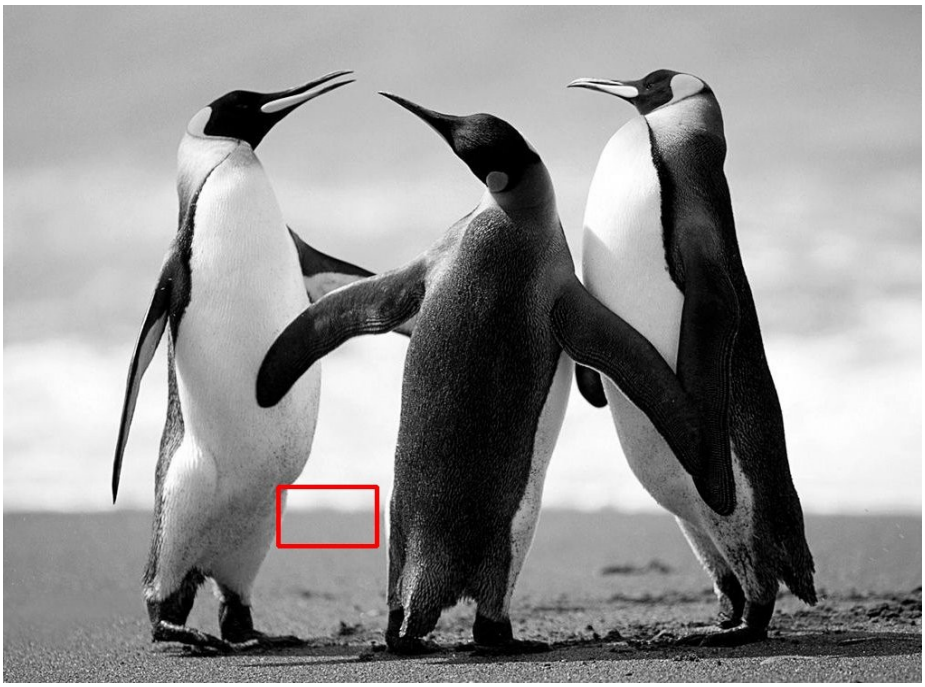
El programa fue probado, en sus variantes GPU y CPU, con tres imágenes distintas, para comparar los resultados. Cada par de imágenes está adjuntada en la entrega.

En primer término se utilizaron las imágenes provistas por la cátedra, y luego algunas imágenes propias:

Imágenes 1



Imágenes 2



Imágenes 3

(La imagen template es intencionalmente muy pequeña, para testear si el matching aún era correcto en estas condiciones)

Como puede verse se tomaron tres imágenes distintas; la primera como prueba básica, la segunda con una imagen y un área de matching un tanto más grandes, y la tercera con un área de matching ínfima, para testear las diferencias de tiempos y la efectividad del algoritmo (que funcionó bien en todos los casos).

Los resultados, en tiempo, de ambas versiones (CPU y GPU), en segundos, se presentan a continuación. Los resultados son promedios estimativos medidos en base a varias ejecuciones de cada algoritmo.

Set de imágenes	**CPU**	**GPU**
1	45	0.53
2	48	0.54
3	0.89	0.52

Se observa que, para volúmenes de datos más grandes, la mejora en el algoritmo corrido en GPU es enorme (en el orden de casi 100 veces más rápido para las imágenes 1 y 2). Esto se debe a la ejecución en paralelo de threads que calculan la semejanza en cada sector, contra la ejecución secuencial de los bucles en el algoritmo por CPU, además de la velocidad de la GPU para realizar las operaciones matemáticas usadas para calcular el promedio de semejanza

Dado que el algoritmo no recorre cada píxel de toda la imagen, sino que la recorre hasta los bordes teniendo el cuenta el tamaño de la imagen template, se hace cada vez más lento a medida que se aumenta el tamaño de la imagen template hasta cierto punto; a partir de allí vuelve a comenzar a ser más rápido, porque en realidad está recorriendo un rango menor en el bucle “exterior” del algoritmo. Es decir, el algoritmo en CPU será bastante más lento a medida que la cantidad de iteraciones en ambos bucles, sumadas, sea mayor; mientras que en el algoritmo en GPU, no se notará un salto tan grande. Esto ocurre en un punto de “equilibrio” entre los tamaños de ambas imágenes.

También debido a esto se notan resultados similares en la Imagen 3: la imagen template es muy pequeña, con lo cual el bucle “interior” del algoritmo constará de unas pocas iteraciones, y terminará siendo igual una recorrida de cada píxel de la imagen base, multiplicada por el (muy pequeño) número de píxeles de la imagen template. Aquí se aproxima bastante a la performance de la versión de GPU, que siempre tiene un overhead por la carga de datos. Aún así, la versión GPU la supera, por la ejecución concurrente y su capacidad para operaciones matemáticas.