Práctica 3

Juan Camilo Acosta Rojas, Simón Aparicio Bocanegra, Jhon Sebastián Rojas Rodríguez,

***Resumen*—La computación paralela es una importante herramienta disponible para los diseñadores de aplicaciones para el uso más eficiente de los recursos computacionales disponibles. En el caso de estudio a trabajar en la materia, algoritmos de procesamiento de imágenes, gracias a la disposición matricial de los datos permiten la paralelización de algoritmos de convolución usados. Filtros convolucionales pueden ser aplicados en diversos campos como: detección de bordes, contraste, eliminación de ruido, entre muchas otras. Poder distribuir los subprocesos mediante distintos métodos de paralelización es una solución efectiva y rápida para solucionar dichos problemas.**

***Index Terms*—proceso, paralelizar, imagen.**

I. Introducción

La computación paralela ofrece grandes oportunidades en campos como la congestión de programas en la nube, optimización de procesos, escalabilidad en las páginas web, ejecución de algoritmos complejos. La computación paralela consiste en la división de procesos y su ejecución simultanea para un mejor aprovechamiento de los recursos computacionales. La aplicación de filtros convolucionales nos ofrecen un problema que puede ser dividido y optimizado usando la teoría de la computación paralela para obtener tiempo total de ejecución menor. El objetivo de este trabajo es diseñar, implementar y analizar un algoritmo que aplique un filtro convolucional sobre una imagen y posteriormente paralelizar la ejecución del mismo y comparar los tiempos de procesamientos finales.

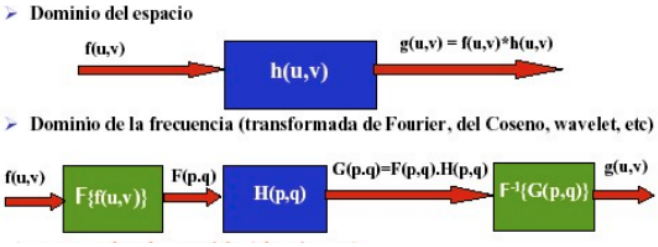
II. Procesamiento de imágenes

Los métodos de procesamiento se pueden dividir en:

- Métodos del espacio: Trabajan directamente sobre el arreglo de píxeles de entrada, es decir, se modifican directamente los píxeles de la imagen. Se trata de algoritmos locales que transforman o bien el valor de cada píxel tomado de manera individual o bien el de un pequeño conjunto de ellos.

- Métodos de frecuencia: Son más costosos en términos computacionales y se utilizan métodos como transformaciones de Fourier.

Estos dos métodos no son mutuamente excluyentes, de hecho, algunos métodos provienen de la combinación de ellos. Un caso de aplicación de estos métodos son los filtros y las convoluciones. Existen además dos tipos de transformaciones: transformaciones globales, donde cada píxel de salida depende de un píxel de entrada, el resultado no varía de acuerdo a la vecindad de los píxeles, si son permutados aleatoriamente o si son reordenados; y transformaciones locales, donde el valor de cada píxel depende de la vecindad del mismo.



[1]

Figura 1. Procesamiento de imágenes

- De manera global:
  $R(x,y) := f(A(x,y))$ ó $R(x,y) := f(A(x,y), B(x,y))$
- De manera local:
  $R(x,y) := f(A(x-k, y-k), ..., A(x,y), ..., A(x+k, y+k))$

Mediante estos procedimientos ya sean globales o locales, se puede modificar una imagen.
Dentro de las convoluciones que se pueden aplicar en el procesamiento de imágenes las convoluciones discretas, las más usadas, son una combinación lineal de los valores de los píxeles vecinos de la imagen mediante una matriz de coeficientes que es conocida como la máscara o el kernel de convolución. Una operación de convolución modifica además el tamaño de la imagen reduciéndolo, razón por la cual es común encontrar padding, donde se asigna 0 a los píxeles que no pueden ser calculados.
Dentro de los efectos destacados que se pueden obtener con filtros convolucionales se encuentran:

- Suavizado: Se entiende como la difuminación de la imagen, reducir los contrastes abruptos dentro de esta.
- Perfilado: Resaltar los contrastes, lo contrario al suavizado.
- Bordes: Proceso de detección de variaciones bruscas dentro del arreglo de píxeles de la imagen.
- Detección: Detección de cierto tipos de rasgos característicos en la imagen como esquinas, segmentos, entre otros.

Para el proceso de detección de bordes en una imagen existen diferentes filtros que siguen un comportamiento estadístico específico. [1].

III. Filtros Sobel

EL filtro de Sobel es uno de los filtros de convolución de imágenes basados en los filtros espaciales de realce, los cuales consisten en resaltar características de las imágenes que hayan podido quedar emborronadas. Estos filtros están asociados a la detección de lados o bordes. El filtro Sobel detecta bordes horizontales y verticales separadamente sobre

una imagen. El origen del filtro proviene del cálculo de un operador local de derivación ya que un píxel pertenece a un borde si se produce un cambio brusco entre niveles de grises con sus vecinos. Mientras más brusco el cambio es más fácil detectar el borde [2].
La derivada de una función digital se define en términos de las variaciones entre píxeles adyacentes. Para ello, la primera derivada debe ser 0 en zonas de intensidad constante y distinta a 0 en zonas de variaciones, además, la segunda derivada debe ser 0 en zonas de intensidad constante y a lo largo de rampas con pendiente constante y distinta de 0 si se presentan variaciones, escalones y el comienzo o fin de una rampa. Para el caso del operador Sobel, se suelen usar dos máscaras para modelizar el gradiente que se llaman operadores de Sobel en donde en la matriz de coeficientes tienen más peso los píxeles situados en la posición vertical y horizontal respecto a los píxeles que están situados en la diagonal, lo que lo hace menos sensible al ruido. El gradiente digital obtenido de acuerdo al concepto de gradiente digital es [3]:

$$\nabla f(x) = \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \tag{1}$$

Para el caso del filtro Sobel se tiene que:

$$G_x = (z_3 + 2z_6 + z_9) - (z_1 + 2z_4 + z_7) \tag{2}$$

$$G_y = (z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3) \tag{3}$$

[4]

IV. Diseño e implementación

Para esta segunda entrega se planeó el diseño de un programa secuencial paralelizable que implementa el filtro Sobel sobre una imagen. Para poder procesar las imágenes se utilizó la librería **stbimage.h**, que es una librería para tratamiento de imágenes en el lenguaje de programación C y C++. Esta librería permite hacer el cargue de una imagen en una matriz de enteros sin signo, esta matriz del tamaño de la imagen y del número de canales de la misma, que es 1 para imagenes en escala de grises y 3 para imagenes a color. Los kernels escogidos son los siguientes:

$$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \tag{4}$$

$$\begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & 2 & -1 \end{pmatrix} \tag{5}$$

$$\begin{pmatrix} 0 & 0 & 0 & -1 & -1 & -1 & 0 & 0 & 0 \\ 0 & -2 & -3 & -3 & -3 & -3 & -3 & -2 & 0 \\ 0 & -3 & -2 & -1 & -1 & -1 & -2 & -3 & 0 \\ -1 & -3 & -1 & 9 & 9 & 9 & -1 & -3 & -1 \\ -1 & -3 & -1 & 9 & 19 & 9 & -1 & -3 & -1 \\ -1 & -3 & -1 & 9 & 9 & 9 & -1 & -3 & -1 \\ 0 & -3 & -2 & -1 & -1 & -1 & -2 & -3 & 0 \\ 0 & -2 & -3 & -3 & -3 & -3 & -3 & -2 & 0 \\ 0 & 0 & 0 & -1 & -1 & -1 & 0 & 0 & 0 \end{pmatrix} \tag{6}$$

Estos filtros siguen el comportamiento de los filtros Sobel para la detección de bordes de una imagen. En este caso, se desea utilizar un producto de matrices para realizar la convolución con cada uno de los filtros mostrados.

El algoritmo usado consiste en la iteración sobre subselecciones de píxeles de la imagen, su multiplicación convolucional con el kernel

Para ello se va a implementar una convolución por cada canal de los píxeles de la imagen de entrada, es decir, un procedimiento de multiplicación de matrices por cada canal e insertando el valor de cada convolución de cada píxel en una reserva de memoria del mismo tamaño de la imagen de entrada. Posteriormente, se escribe la imagen que obtiene la convolución y, por último, se carga.

Para esta entrega, se va a emplear la paralelización basada en la arquitectura de **CUDA** (Compute Unified Device Architecture) mediante el CUDA Toolkit que se implementó en una GPU Nvidia y en un notebook de Google Colaboratory, en donde se emplea la ejecución mediante la GPU propuesta por Google para este tipo de instancias.

El algoritmo consiste en la asignación de una fila a un hilo para el calculo de sus valores. Así, el número de hilos para cada ejecución depende unicamente del tamaño de la imagen y es constante entonces para la imagen. Cada hilo itera repetidamente sobre una sección del tamaño del kernel a lo largo de la imagen por cada pixel. Se decidió guardar en cada ejecución la matriz del kernel en la memoria compartida (shared mem) de cada multiprocesador. En la ejecución, siendo el número de hilos para cada resolución constante, el usuario puede cambiar el número de hilos por bloque y así cambiar el número de bloques a ejecutar, siendo estas dos variables inversamente proporcionales.

V. HARDWARE

CUDA Device Query (Runtime API) version (CUDART static linking)
Detected 1 CUDA Capable device(s)
Device 0: ”NVIDIA GeForce GTX 1050”
CUDA Driver Version / Runtime Version 11.5 / 11.5
CUDA Capability Major/Minor version number: 6.1
Total amount of global memory: 4040 MBytes (4236312576 bytes)
(005) Multiprocessors, (128) CUDA Cores/MP: 640 CUDA Cores
GPU Max Clock rate: 1493 MHz (1.49 GHz)
Memory Clock rate: 3504 Mhz
Memory Bus Width: 128-bit
L2 Cache Size: ¡ 524288 bytes
Maximum Texture Dimension Size (x,y,z) 1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
Maximum Layered 1D Texture Size, (num) layers 1D=(32768), 2048 layers
Maximum Layered 2D Texture Size, (num) layers 2D=(32768, 32768), 2048 layers
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total shared memory per multiprocessor: 98304 bytes
Total number of registers available per block: 65536
Warp size: 32
Maximum number of threads per multiprocessor: 2048
Maximum number of threads per block: 1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)
Maximum memory pitch: 2147483647 bytes
Texture alignment: 512 bytes
Concurrent copy and kernel execution: Yes with 2 copy engine(s)
Run time limit on kernels: Yes
Integrated GPU sharing Host Memory: No
Support host page-locked memory mapping: Yes
Alignment requirement for Surfaces: Yes
Device has ECC support: Disabled
Device supports Unified Addressing (UVA): Yes
Device supports Managed Memory: Yes
Device supports Compute Preemption: Yes
Supports Cooperative Kernel Launch: Yes
Supports MultiDevice Co-op Kernel Launch: Yes
Device PCI Domain ID / Bus ID / location ID: 0 / 1 / 0
Compute Mode:
Default (multiple host threads can use ::cudaSetDevice() with device simultaneously)
deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.5, CUDA Runtime Version = 11.5, NumDevs = 1
Result = PASS

VI. ANÁLISIS Y RESULTADOS

Para poder medir la eficiencia del programa, se decidieron realizar tres experimentos con cada uno de los kernel mencionado anteriormente con imágenes de distinta resolución (720p, 1080p y 4k), en donde para cada experimento se midieron los tiempos de repuesta de acuerdo al número de bloques y al número de hilos manejado. Además de esto, se calcula SppedUp para medir la aceleración del algoritmo cuando este es paralelizado mediante la arquitectura CUDA.

Imagen	Bloques	Hilos	Tiempo
720p	45	16	0.258667
720p	23	32	0.289817
720p	12	64	0.247656
720p	6	128	0.212225
720p	3	256	0.213413
720p	2	512	0.250731
1080p	67	16	0.348307
1080p	34	32	0.361214
1080p	17	64	0.3518
1080p	9	128	0.370923
1080p	5	256	0.343643
1080p	3	512	0.407788
4k	135	16	1.728446
4k	68	32	1.682802
4k	34	64	1.738224
4k	17	128	1.783708
4k	9	256	1.739336
4k	5	512	1.776671
8k	270	16	3.614216
8k	135	32	5.054428
8k	68	64	4.925712
8k	34	128	5.109668
8k	17	256	5.093284
8k	9	512	4.873379
12k	351	16	13.982807
12k	176	32	19.028077
12k	88	64	21.087436
12k	44	128	20.329245
12k	22	256	19.549508
12k	11	512	20.206168

Cuadro I
TABLA 1. TIEMPOS DE RESPUESTA DEL ALGORITMO DE ACUERDO A LA IMAGEN, EL NÚMERO DE BLOQUES Y EL NÚMERO DE HILOS CON EL KERNEL DE 9X9

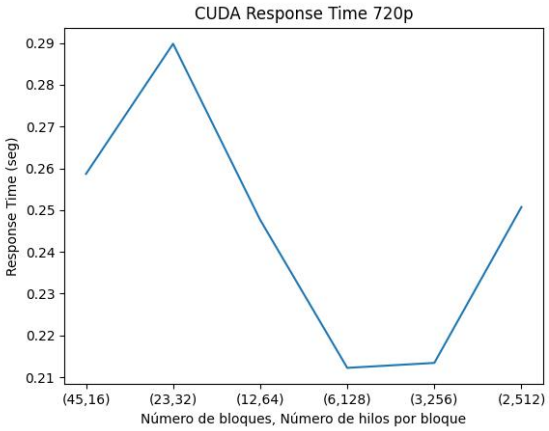


Figura 2. Tiempos de respuesta para el kernel 9x9 para una resolución de 720p variando el número de bloques y de hilos por bloque.

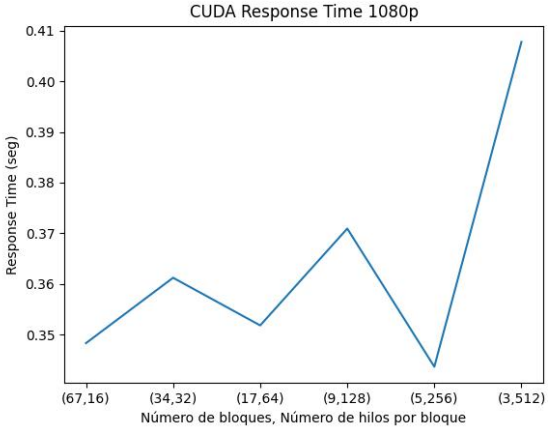


Figura 3. Tiempos de respuesta para el kernel 9x9 para una resolución de 1080p variando el número de bloques y de hilos por bloque.

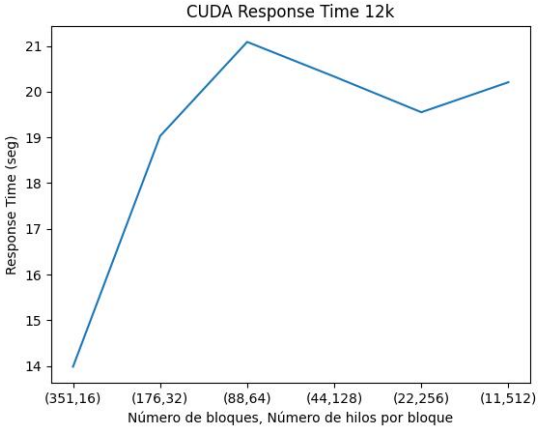


Figura 6. Tiempos de respuesta para el kernel 9x9 para una resolución de 12k variando el número de bloques y de hilos por bloque.

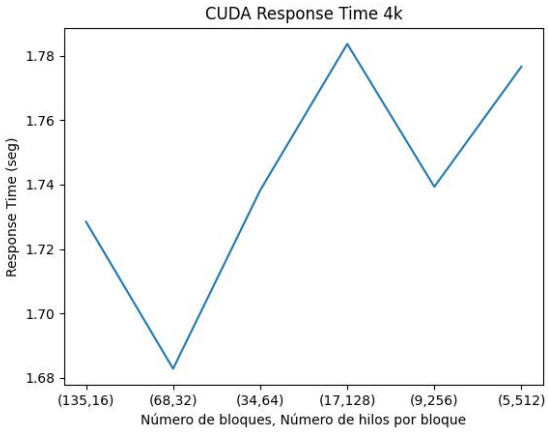


Figura 4. Tiempos de respuesta para el kernel 9x9 para una resolución de 4k variando el número de bloques y de hilos por bloque.

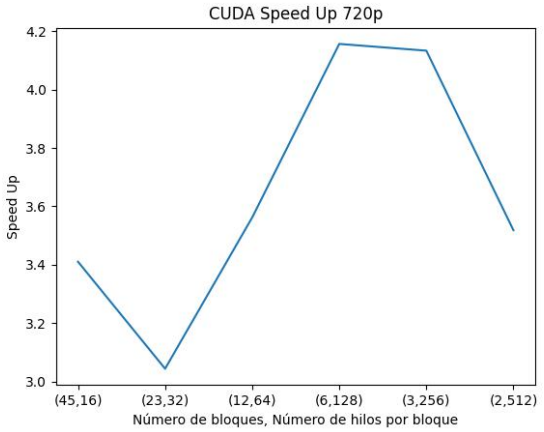


Figura 7. Spped Up para el kernel 9x9 para una resolución de 720p variando el número de bloques y de hilos por bloque.

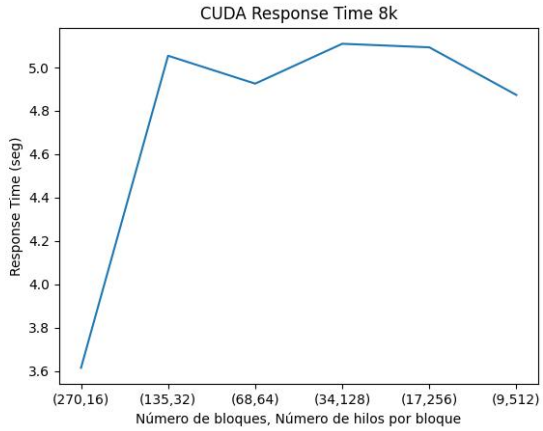


Figura 5. Tiempos de respuesta para el kernel 9x9 para una resolución de 8k variando el número de bloques y de hilos por bloque.

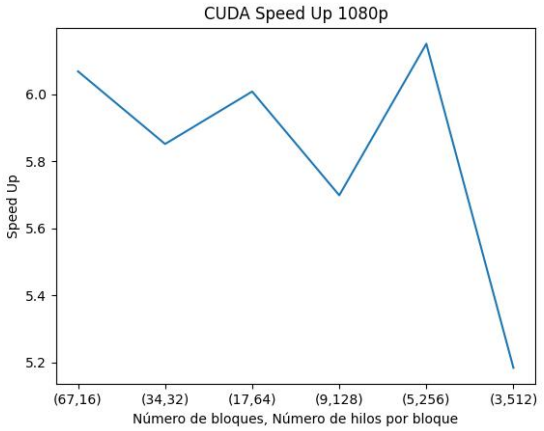


Figura 8. Spped Up para el kernel 9x9 para una resolución de 1080p variando el número de bloques y de hilos por bloque.

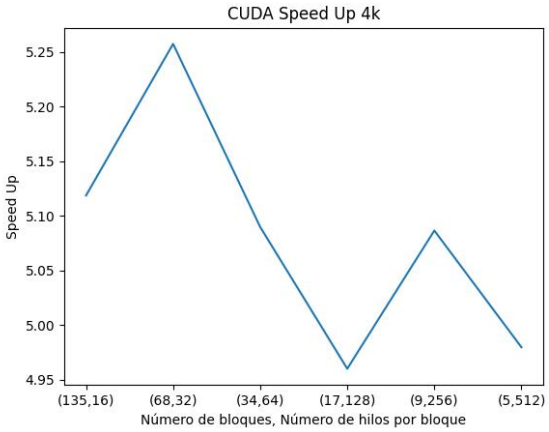


Figura 9. Spped Up para el kernel 9x9 para una resolución de 4k variando el número de bloques y de hilos por bloque.

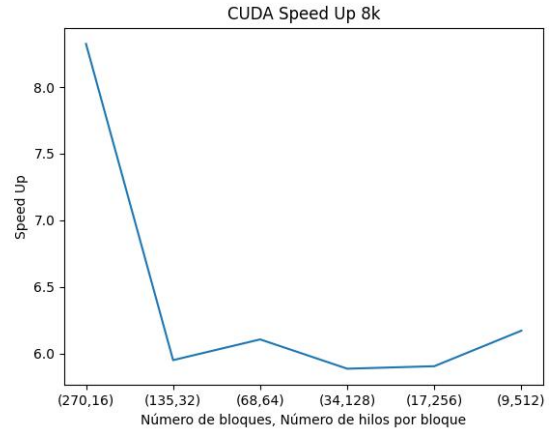


Figura 10. Spped Up para el kernel 9x9 para una resolución de 8k variando el número de bloques y de hilos por bloque.

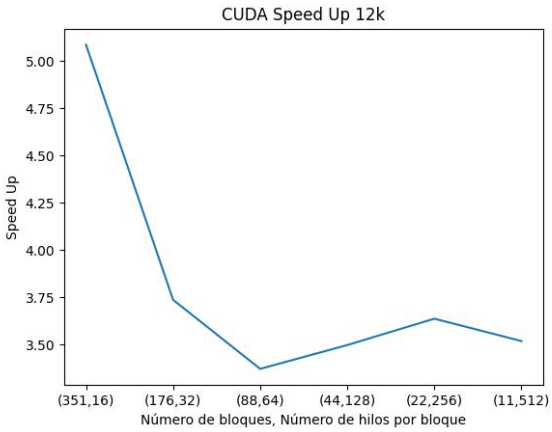


Figura 11. Spped Up para el kernel 9x9 para una resolución de 12k variando el número de bloques y de hilos por bloque.

Podemos notar en general pequeñas diferencias al cambiar el número de núcleos por bloque y así el número de bloques a ejecutar, sin embargo para varias de las resoluciones parecen ser menores lo tiempos de respuesta para un número menor de hilos por bloque y un número mayor de bloques.

VII. Conclusiones

- Los algoritmos de procesamiento de imágenes cumplen un papel muy importante en varios ámbitos de la ciencia y la investigación, y poder optimizar sus procesos para obtener una respuesta más rápida es un logro importante debido a las diferentes complejidades que tiene dicho procesamiento.
- La librería OMP se pueden paralizar diferentes procesos de diferentes maneras, para el caso de multiplicación de matrices, dividiendo en diferentes secciones se puede lograr un mejor resultado en términos de tiempos de respuesta y Speed Up del proceso.
- Con respecto al programa secuencial elaborado en la primera práctica, se observa una mejoría considerable usado métodos de paralelización, por lo que la implementación de métodos de la computación paralela son efectivos en este tipo de operaciones, sin embargo para ciertas tareas y ciertas formas de paralización no lo es, se debe tener un análisis previo antes de poder considerar un proceso como paralelizable.
- Es posible ejecutar el algoritmo de filtro Sobel haciendo uso de una GPU junto con la arquitectura CUDA de Nvidia, sin embargo se debe seleccionar el número de bloques y número de hilos por bloque necesarios para sacarle el mayor provecho posible a este hardware.
- Variar el número de hilos por bloque, no parece generar ninguna mejora en los tiempos de respuesta, esto puede ser debido a la implementación del filtro en la que se le asigna una fila de la imagen a cada hilo, por lo que en todas las ejecuciones se tienen en

Referencias

[1] B.E.R, Jimena Olivares Montiel, Çonvolución y filtrado", LAPI [Online]. Available: http://lapi.fi-p.unam.mx/wp-content/uploads/6-Filtros-y-morfologia.pdf
[2] "Filtros de detectar bordes: Sobel"[Online]. Available: https://docs.gimp.org/2.8/es/plug-in-sobel.html
[3] "Tema 3: Filtros"[Online]. Available: http://grupo.us.es/gtocoma/pid/tema3-2.pdf
[4] I.P. Nacional, .Extracción de bordes: Operadores de Sobel, Prewit y Roberts", Boletín UPIITA [Online]. Available: http://grupo.us.es/gtocoma/pid/tema3-2.pdf