Procesamiento puntual de imágenes digitales usando cómputo paralelo

Eric Olmedo González Jorge De La Calleja Antonio Benitez Ma Auxilio Medina
Departamento de Posgrado
Universidad Politécnica de Puebla

***Resumen*—En el presente trabajo se muestra el uso del cómputo paralelo aplicado al tratamiento de imágenes digitales, específicamente al filtro de escala de grises. Se utilizó *CUDA* como herramienta de programación paralela para realizar el procesamiento sobre una GPU de imágenes digitales en varias resoluciones con el objetivo de aprovechar cada uno de sus núcleos para cálculo masivo de información. La técnica utilizada aplica una transformación sobre cada pixel en cada núcleo de forma concurrente en lugar de secuencial. Obteniendo como resultado tiempos de ejecución menores en la GPU respecto a la función de escala de grises implementada por la biblioteca *OpenCV* que se ejecuta en un CPU.**

***palabras clave*—procesamiento digital de imágenes, procesamiento puntual, escala de grises, cómputo paralelo, cuda.**

I. Introducción

El procesamiento digital se ha convertido en una área de investigación aplicada desde la fotografía profesional hasta campos como la astronomía, meteorología, visión por computadora, imágenes médicas, entre otros. El objetivo del procesamiento digital de imágenes es mejorar la información pictórica, posteriormente se aplicó a otras tareas como clasificación de imágenes, extracción de características, reconocimiento de patrones, por nombrar algunas.

El procesamiento de imágenes usualmente es una tarea pesada en trabajo y tiempo, por ejemplo, para el procesamiento puntual de una imagen de 1024×1024 pixeles en escala de grises, se requieren más de un millón de operaciones, y si además el trabajo se aplica a una imagen en color, las operaciones se multiplican por el número de canales.

En años recientes las tarjetas de video o unidades gráficas de procesamiento (GPU) se convirtieron en una herramienta de procesamiento de grandes cantidades de información (en el número de millones de datos) en paralelo. NVIDIA desarrolló la arquitectura CUDA que agrupa los núcleos de una GPU en un vector que puede ser programado para reducir el tiempo de procesamiento de grandes cantidades de datos.

El uso de cómputo paralelo usando una GPU comenzó hace varios años, por ejemplo, en 2004 Fung y Mann [1] propusieron una nueva arquitectura usando varias GPUs para el procesamiento de imágenes y visión por computadora, como resultado obtuvieron un significativo incremento de velocidad comparado con la implementación en CPU. En 2006 Farrugia et al [2] desarrollaron la biblioteca GPUCV para acelerar el procesamiento de imágenes usando GPUs; observaron que el procesamiento con dicha biblioteca es entre 1.2 a 18 veces más rápido que la versión en OpenCV. Con la aparición de CUDA, el desarrollo se centró en el diseño de algoritmos dejando a un lado la tarea de cómo hacer funcionar un GPU para el procesamiento de la información. Para 2007 Sham et al [3] presentaron un método eficiente para el cálculo de información mutua (MI) entre imágenes. Ellos mejoraron la eficiencia de los cálculos en un factor de 25 comparado con la implementación en CPU. Fung and Mann [4] en 2008 usaron CUDA para "ayudar a convertir imágenes en números", es decir, visión por computadora. Ellos obtuvieron una mejora de velocidad de 9.8 hasta 21 veces superior respecto a la versión de CPU. También en 2008 Zhiyi et al [5] mejoraron el tiempo de ejecución de algunos filtros como la ecualización de histograma, la codificación y decodificación DCT y algoritmos de detección de bordes obteniendo tiempos de 8 hasta 200 veces más rápidos que la de CPU. Tarabalka et al [6] en 2009 usaron GPUs para el procesamiento en tiempo real de grandes volúmenes de datos grabados por una imagen hiperespectral, reportaron que su implementación con GPUs se ejecuta entre 10 a 100 veces más rápido.

Este artículo presenta un método para el procesamiento digital de imágenes usando cómputo paralelo con el fin de reducir los tiempos de ejecución. Se probó este método con imágenes de diferentes resoluciones para la escala de grises teniendo como resultado una ejecución más rápida que la función equivalente de la biblioteca OpenCV.

Las secciones subsecuentes se organizan como sigue: en la sección 2, 3 y 4 se presenta una breve explicación sobre procesamiento digital de imágenes, cómputo paralelo, CUDA y OpenCV. La sección 5 describe el método propuesto, mientras que los resultados de los experimentos se presentan en la sección 6. Por último, las conclusiones y el trabajo a futuro se encuentran en la sección 7.

II. Procesamiento digital de imágenes

El procesamiento digital de imágenes, de acuerdo a [7], consiste en aplicar una función que transforma una imagen bidimensional mediante una computadora digital. Otros autores como Crane [8] definen esa tarea como una ciencia que manipula una imagen digital y cubre un amplio conjunto de técnicas ya sea para resaltar o distorsionar la imagen.

Una imagen digital es un conjunto de bits que representan algo, este conjunto es obtenido a través de un sensor de visión y es transformada en un formato *digital*. Formalmente, una imagen digital se define como una función bidimensional $f(x, y)$ donde x e y son coordenadas de un plano y f es la intensidad en algún punto del plano. Cuan-

do x, y y f son cantidades finitas y discretas, se dice que la imagen es digital. Cada elemento de una imagen se llama *elemento de imagen* o *pixel* [9].

Un *espacio de colores* es una forma de representar colores y su relación entre ellos. Las personas tienen una visión tri-cromática, es decir, se conforma de tres receptores que reaccionan a los colores rojo(R), verde(G) y azul(B). El espacio de colores *RGB* funciona de la misma forma, tienen tres canales para cada color [10].

Una imagen digital en color es una matriz de pixeles, donde cada elemento está integrado por tres valores en el rango $[0, 255]$. La combinación de los canales definen un color y es posible obtener hasta 16.8 millones de colores aproximadamente $(256 \times 256 \times 256)$.

Existen varios tipos de transformaciones o procesamientos que pueden ser aplicados a una imagen digital tales como operaciones orientadas al punto, orientadas a la región u operaciones geométricas entre imágenes. En el presente trabajo se abordará del procesamiento orientado al punto, debido a que son los filtros más sencillos de implementar en el procesamiento de imágenes.

A. *Procesamiento puntual*

En el *procesamiento puntual* una transformación modifica un pixel sobre un solo canal.

Sea $f(x, y)$ el valor de un pixel en las coordenadas x e y, $g(x, y)$ una transformación sobre $f(x, y)$ y T una función. Entonces T mapea $f(x, y)$ a $g(x, y)$ donde un solo pixel es afectado en la coordenada (x, y) como se muestra en la ecuación (1).

$$g(x, y) = T[f(x, y)] \tag{1}$$

Entre estas funciones se encuentran: el negativo, la escala de grises, aclarado, obscurecimiento, reducción o aumento de brillo, reducción o aumento de contraste, entre otras.

B. *Filtro de escala de grises*

Una imagen en escala de grises es resultado de tomar las intensidades de luz en lugar de colores, algunas personas llaman a este filtro incorrectamente una imagen en blanco y negro [11]. Una forma de obtener esto es promediando las intensidades de los tres canales como se muestra en la ecuación (2).

$$GI(x, y) = \frac{Rojo(x, y) + Verde(x, y) + Azul(x, y)}{3} \tag{2}$$

donde GI es la imagen en escala de grises.

Pratt [12] sugiere utilizar la ecuación (3) debido a la sensibilidad del ojo a los colores, es decir, la visión tiene un orden en el grado de sensibilidad a los colores: primero verde, después rojo y por último el azul. Por lo que al utilizar la ecuación (3) el color verde tendrá un brillo mayor respecto a los demás, a esta función se le llama *luminancia* [12].

$$Gris(x, y) = 0{,}3 \times Rojo(x, y) + 0{,}59 \times Verde(x, y) + 0{,}11 \times Azul(x, y) \tag{3}$$

En la Figura 1. se muestra el resultado de aplicar la transformación a una imagen en colores *(a)* obteniendo una imagen en escala de grises *(b)*. En la siguiente sección se presentará el cómputo paralelo y algunas de sus características.

(a) (b)

Figura 1. Transformación de una imagen a color en escala de grises.

III. Cómputo paralelo

El cómputo paralelo es una opción para resolver problemas que requieren grandes cantidades de procesamiento o manejo de grandes cantidades de información en un tiempo (de acuerdo a cada criterio) "aceptable". En el procesamiento paralelo, un programa crea múltiples tareas que cooperan para resolver un problema [13]. La idea principal es dividir un problema en tareas más sencillas y resolubles de forma concurrente, de tal manera que el tiempo total pueda ser dividido entre el número de tareas en el mejor de los casos.

Es importante mencionar que no se puede aplicar el procesamiento paralelo a todos los problemas, es decir, no todos los problemas son paralelizables. Un programa paralelo debe contar con algunas características para un correcto y eficiente funcionamiento, de lo contrario es posible que el tiempo de ejecución o el funcionamiento no sea el esperado. Entre estas características se encuentran las siguientes [14]:

- *Granularidad.*- Se define como el número de unidades básicas y se clasifica en:
  Grano grueso.- Pocas tareas de cómputo más intenso.
  Grano fino.- Una gran cantidad de piezas pequeñas y de cómputo menos intenso.
- El tipo de paralelismo:
  Explícito.- El algoritmo incluye las instrucciones necesarias para especificar cuáles procesos son paralelos y como deben ejecutarse.
  Implícito.- El compilador tiene la tarea de insertar las instrucciones necesarias para ejecutar el programa sobre una computadora paralela.
- *Sincronización.*- Esta característica evita que dos o más procesos se traslapen.
- *Latencia.*- Es el tiempo de transición de información desde la solicitud hasta la recepción.
- *Escalabilidad.*- Se define como la capacidad de un algoritmo de mantener su eficiencia al aumentar el número de procesadores y el tamaño del problema en la misma proporción [15].

- *Aceleración* y *eficiencia* son métricas que permiten apreciar la calidad de una implementación paralela.

En la siguiente sección se muestra la arquitectura paralela CUDA y algunas características tanto de su diseño cómo de su entorno de programación.

A. CUDA

Desde hace 10 años se establecieron dos enfoques respecto al diseño de microprocesadores; los *multicores* dirigidos a mantener la velocidad de ejecución de programas secuenciales cuando se mueven entre núcleos (cores) y los *many-cores* que se encargan de la ejecución de aplicaciones paralelas [16].

Entre los *many-cores* se encuentran las tarjetas gráficas o unidades gráficas de procesamiento (GPU). La Figura 2. muestra que los GPUs tienen mejor rendimiento que los CPU's. En 2009 la diferencia entre GPUs y CPUs en operaciones de punto flotante fue aproximadamente de 10 a 1, en otras palabras, una GPU alcanzó 1 teraflop (1,000 gigaflops[1]) en tanto el CPU alcanzó solamente 100 gigaflops de rendimiento [16]. Esta diferencia radica principalmente en la filosofía del diseño de los procesadores.

La idea de usar una GPU para el procesamiento numérico intenso motivó la creación el diseño de *CUDA*® (Arquitectura de Dispositivo de Cómputo Unificado) del inglés "Compute Unified Device Architecture", un modelo de programación para la ejecución de aplicaciones en un CPU y una GPU [16].

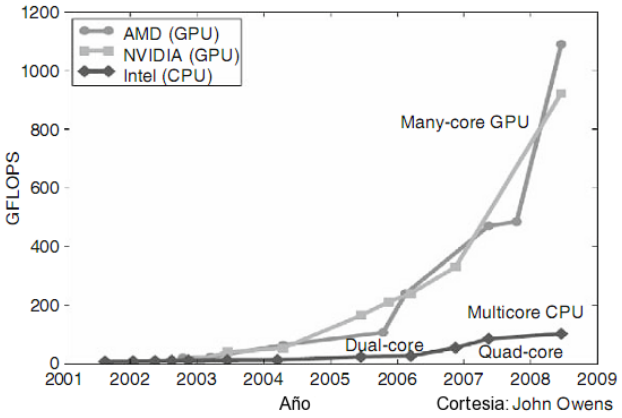

Figura 2. Evolución del rendimiento de CPU vs GPU

En la Figura 3. se muestra la arquitectura de una GPU *CUDA-capable*. Ésta se organiza en módulos llamados "Streaming Multiprocessors" (SMs), dos SMs forman un bloque. Los SMs están formados por "Streaming Processors" (SPs) que comparten un control lógico y un espacio caché de instrucciones, el total de SPs dependen del modelo de GPU. La GPU tiene un espacio de memoria global compartida por todos los SMs donde el acceso es secuencial, a diferencia del aleatorio de un CPU. Los SPs pueden ejecutar múltiples hilos por aplicación (pueden ser miles de hilos) a diferencia de los CPUs que sólo pueden tomar 2 ó 4 hilos por núcleo. Por ejemplo, el chip G80 tiene 128 SPs agrupados en 16 SMs y cada uno puede ejecutar hasta 768 hilos, como resultado se pueden ejecutar más de 12,000 hilos en el chip.

La programación paralela usando *CUDA* es *explícita* y de *grano fino*, es decir, es necesario diseñar cómo se dividen las tareas para ser ejecutadas en paralelo y cómo éstas deben comunicarse. Las tareas deben ser lo más sencillas posible, es decir, operaciones mínimas que no pueden descomponerse en tareas más sencillas. En un programa *CUDA*, una función *kernel* indica el código que será ejecutado por los hilos durante la fase paralela. Estas funciones son identificadas como *host*, *device* y *global* [16]. La primera se refiere a la sección del programa que se ejecutará solamente en el CPU, la segunda solamente en la GPU y la tercera son las secciones donde el CPU y GPU podrán comunicarse.

En la sección IV se proporciona una breve descripción de *OpenCV*, una biblioteca para el procesamiento de imágenes y aprendizaje automático.

IV. OpenCV

OpenCV (Open Source Computer Vision) es una biblioteca de código abierto desarrollada inicialmente por Intel, la cual proporciona funciones para crear aplicaciones de visión por computadora y aprendizaje automático en tiempo real. En la Figura 4. se muestran los tres módulos que conforman *OpenCV*, uno para el procesamiento de imágenes, otro para aprendizaje automático y un último para manejar imágenes, videos y funciones para crear interfaces gráficas de usuario. La biblioteca está programada en C y C++, puede ejecutarse en ambientes Linux, Windows y Mac OS X. Debido a que fue creada por Intel®, es posible obtener códigos optimizados usando la biblioteca "Integrated Performance Primitives" (IPP), formada por rutinas mejoradas de bajo nivel usadas en varios algoritmos [17].

La biblioteca está conformada por más de 500 funciones de visión. Existen funciones para el procesamiento de imágenes digitales con filtros y máscaras que pueden aplicarse a una imagen para mejorar la calidad o encontrar información que no es visible o disponible en primera instancia. Estas funciones son utilizadas en varias áreas como la inspección en la producción de una industria, imágenes médicas, seguridad, interfaces de usuario, calibración de cámaras, visión estéreo y robótica, por mencionar algunas.

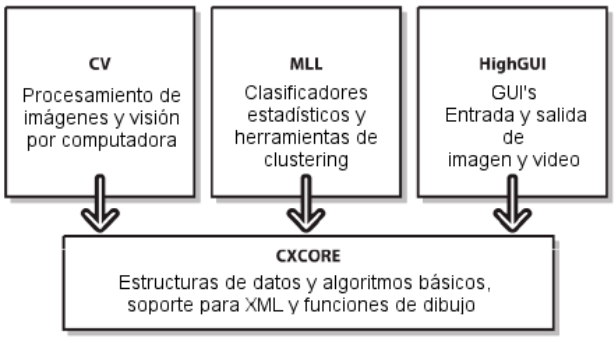

Figura 4. Diagrama de la biblioteca *OpenCV*.

[1] Un gigaflop son mil millones de operaciones de punto flotante

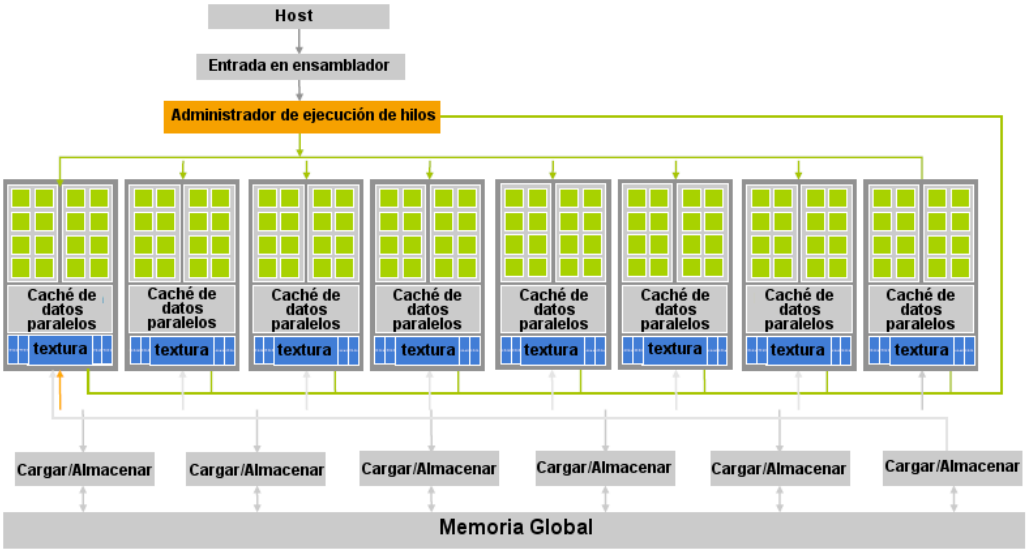

Figura 3. Arquitectura de un GPU *"CUDA-capable"*.

V. Implementación de la transformación de imágenes en escala de grises mediante cómputo paralelo

Como se mencionó en la sección B: escala de grises del capítulo II, existen 2 ecuaciones que aplican la transformación a escala de grises de una imagen a color. Para este trabajo se decidió utilizar la ecuación (2) y se implementó en un módulo en *CUDA*. En la Tabla 1. se muestra de forma general los pasos a seguir para obtener una imagen en escala de grises usando una GPU.

Algoritmo 1. Algoritmo general para convertir una imagen a color en escala de grises

Entrada: una imagen a color I
Salida: una imagen en escala de grises GI'

1 La imagen I se convierte en un vector I' de enteros donde los tres canales de cada pixel se almacenan consecutivamente.
2 El vector I' se transfiere a la memoria de la GPU.
`/* Las líneas 3, 4 y 5 se ejecutarán en la GPU */`
3 **Para** $i = 0$ *hasta (ancho* $\times$ *alto)* **hacer**
4 $\quad GI[i] = (I'[i\times 3] + I'[i\times 3+1] + I'[i\times 3+2])/3;$
5 **fin-Para**

donde
$GI[i]$ es la imagen en escala de grises en el pixel i
$I'[i\times 3]$, $I'[3\times i+1]$ e $I'[i\times 3+2]$ son los valores rojo, verde y azul respectivamente en el vector I' del pixel i.

6 A continuación el vector GI se transfiere a la memoria RAM.
7 Por último el vector GI se convierte en una imagen GI'.

En la Figura 5. se muestra un diagrama de este proceso que ayudará a comprender los pasos que se encuentran en el Algoritmo 1.

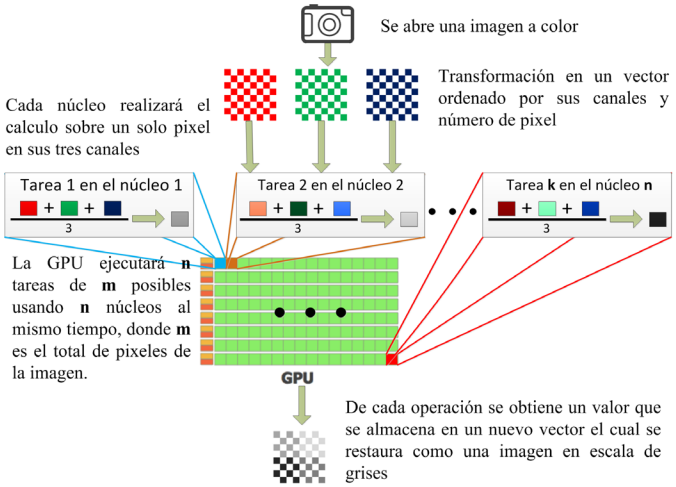

Figura 5. Procedimiento de la escala de grises en una GPU

En las línea 1 del Algoritmo 1. la imagen I es convertida en un vector I' donde los elementos estarán ordenados por el número de pixel y por cada uno de los canales de la imagen (r, g, b). El vector resultante será visto de la siguiente forma:

- $(r_1,\ g_1,\ b_1,\ r_2,\ g_2,\ b_2,\ ...,\ r_n,\ g_n,\ b_n)$

donde r, g y b son los canales rojo, verde y azul respectivamente, y n es el total de pixeles de la imagen, de forma que r_1 es el valor del canal rojo del pixel 1, g_1 del canal verde y b_1 del azul.

En las líneas 3, 4 y 5 se realiza el cálculo de la escala de grises sobre el vector I' usando la ecuación (2), pero será procesado por la GPU, donde el ciclo *for* no se ejecuta de la misma manera. El Algoritmo 2. ilustra cómo se procesan las tareas en paralelo usando *CUDA*.

En la línea 1 del Algoritmo 2. se asigna un identificador a cada tarea que se ejecutará en la GPU, el total de tareas será $(ancho \times alto)$ de la imagen. En *CUDA* se crean hilos

Algoritmo 2. Algoritmo en CUDA para obtener una imagen en escala de grises

Entrada: Vector de una imagen a color *I'*
Salida: Vector de una imagen en escala de grises *GI*

Se asigna un identificador a cada tarea.
1 $i = blockIdx.x \times (blockDim.x \times blockDim.y) + blockDim.x \times threadIdx.y + threadIdx.x$;
2 **Para cada tarea** i **hacer**
3 $\quad GI[i] = (I'[i \times 3] + I'[i \times 3 + 1] + I'[i \times 3 + 2])/3$;
4 **fin-Para cada**

donde
GSC[i] es el pixel i del vector en escala de grises y además es calculado por la tarea i en la GPU
I'[i×3], *I'[i×3+1]* y *I'[i×3+2]* son los canales rojo, verde y azul respectivamente del vector *I'*

que se agrupan en bloques (para tener un mejor control) y éstos son procesados por todos los núcleos.

Para el experimento se utilizó una matriz de bloques de 32×32 y una matriz de hilos de $(alto \times ancho)$ / 1024. Para no repetir una tarea ya realizada, debe asignarse un identificador que corresponde con el número de bloque y número de hilo en el bloque. A continuación se explican las variables de la línea 1 del Algoritmo 2.

$blockIdx.x$ es el índice del bloque actual.
$blockDim.x$ es la dimensión del bloque en el eje x.
$blockDim.y$ es la dimensión del bloque en el eje y.
$threadIdx.x$ es hilo actual en el eje x.
$threadIdx.y$ es hilo actual en el eje y.

Para entender mejor la asignación del identificador del Algoritmo 2. suponga que se utiliza una matriz de bloques de 2×2 y una matriz de hilos de 3×3, si $blockIdx.x = 0$, $threadIdx.x = 0$ y $threadIdx.y = 0$ entonces $i = 0$, es decir, se hace referencia a la primer tarea. Por otro lado, si $blockIdx.x = 2$, $threadIdx.x = 2$ y $threadIdx.y = 1$ entonces $i = 23$, en general se hace referencia a la tarea o hilo 24 (se inicia desde cero), dicha tarea se encuentra en la columna 2 de la fila 1 en la matriz de hilos y además está en el bloque 2 de la matriz de bloques.

La Figura 6 muestra de forma gráfica este concepto. En letra grande y negro los bloques, en verde las filas y columnas de los hilos de cada bloque, en azul el número de hilo en el bloque y en rojo el número de hilo en general. El cuadro amarillo señala la tarea No. 23 que se encuentra en la columna 2 y fila 1 del bloque 2.

En la línea 3 del Algoritmo 2. se realiza la operación de la ecuación (2) para cada uno de elementos del vector *I'* que corresponde a cada uno de los pixeles de la imagen original *I*. Estas operaciones serán enviadas como tareas individuales que ejecutarán los núcleos de la GPU, lo cual *GI[i]* no significa necesariamente que el procesador i está procesando el pixel i, sino que algún procesador ejecuta la tarea i de algún pixel k.

Por último, en la línea 6 y 7 del Algoritmo 1. se realiza la transferencia del vector de la memoria de la GPU a la

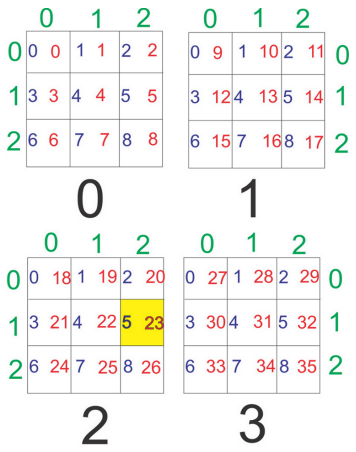

Figura 6. Estructura de un bloque de 3x3 y un grid de 2x2.

RAM del CPU y en ésta se hace la conversión a una imagen para verificar la transformación correcta a escala de grises.

Para visualizar el rendimiento con respecto al tiempo del módulo programado en *CUDA*, se hizo la comparación con la misma función que está implementada en *OpenCV*, la cual se ejecuta sobre el CPU.

VI. Experimentación y resultados

Se implementó un módulo en *CUDA* que aplica la ecuación (2). con la restricción de que la imagen a procesar debe estar previamente almacenada en la memoria de la GPU. En este trabajo se considera sólo el tiempo de procesamiento y no el tiempo de transferencia de información entre CPU y GPU.

Las pruebas se hicieron sobre imágenes de diferentes tamaños, con el fin de verificar si los tiempos de ejecución mantienen la misma proporción al aumentar el tamaño de las imágenes. Los tamaños elegidos son los siguientes:

- 256 × 256
- 512 × 512
- 1024 × 1024
- 1800 × 1400
- 4000 × 3000

El equipo utilizado para este experimento fue una computadora de escritorio con un procesador AMD® Phenom II a 3.2 GHz con 12 GB de memoria RAM, sistema operativo Linux Fedora 14 de 64 bits y la biblioteca OpenCV utilizada fue la versión 2.3. Para el procesamiento con *CUDA* se usó una tarjeta de video GeForce 430 GT con 96 núcleos y 1 GB de RAM DDR3.

La transformación se aplicó a 10 imágenes diferentes de cada resolución, primero con *OpenCV* y posteriormente usando el módulo en *CUDA*. Los tiempos promedio de ejecución obtenidos por ambos métodos son presentados en la Tabla I.

En la Tabla I se puede apreciar que el módulo programado en *CUDA* obtuvo mejores tiempos en todas las resoluciones comparado con la función en OpenCV. En la Figura 7. se puede visualizar esta diferencia, la curva de tiempos referente al módulo en *CUDA* se encuentra de-

Resolución de la imagen	Tiempo (ms) de la función de OpenCV	Tiempo (ms) del módulo en CUDA	Tiempo (ms) de CUDA más transferencia
256 × 256	0.178176	0.102278	0.385833
512 × 512	0.681635	0.386915	1.363168
1024 × 1024	2.847644	1.515747	4.647904
1800 × 1400	6.749038	3.641635	10.651541
4000 × 3000	31.59133	17.269219	50.609980

Tabla I

Tabla comparativa de la transformación a escala de grises usando *OpenCV* y *CUDA*.

bajo de la curva correspondiente a *OpenCV* en todas las resoluciones.

En la Tabla I al compararse los tiempos se puede encontrar que CUDA reducen casi a la mitad los tiempos obtenidos por la función de OpenCV. En la misma tabla existe una tercera columna donde se muestra el tiempo del módulo en CUDA más el tiempo que toma enviar el vector de la memoria RAM a la memoria del GPU. Observando esta última columna, puede caerse en el error de apresurarse a la conclusión que es mejor la implementación en CPU, pero las GPUs están diseñadas para manejar grandes cantidades de información y en este experimento se procesa sólo una imagen a la vez, por lo cual no se aprecia el potencial de una GPU.

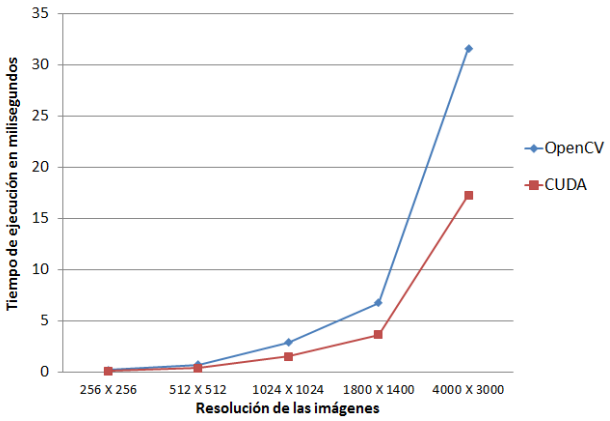

Figura 7. Gráfica de tiempos de ejecución de CPU vs GPU

VII. Conclusiones

En el presente trabajo se mostró el rendimiento respecto al tiempo de procesamiento que tiene una GPU sobre un CPU logrando en la resolución más alta 4000 × 3000 tiempos de 31,59 ms del CPU contra 17,26 ms de la GPU. Estos resultados reflejan el poder de cómputo que existe en una GPU. También se encontró que el tiempo de transferencia de información de la memoria RAM a la memoria de la GPU es mayor que el procesamiento en el CPU, sin embargo, si en lugar de procesar diez imágenes fueran diez mil o diez millones, *CUDA* puede tener mejores tiempos incluyendo la transferencia de información debido a que trabaja de forma asíncrona, es decir, puede estar procesando una imagen mientras otra imagen es transferida a la memoria de la GPU. Otro punto a resaltar es que se utilizó una GPU con 96 núcleos, existe la posibilidad que aumentando el número de núcleos se obtengan mejores resultados, debido a que en el método utilizado para la transformación a escala de grises cada núcleo procesa un pixel a la vez, lo que significa que son procesados al mismo tiempo 96 pixeles, si se utilizara por ejemplo una tarjeta 460 GT que tiene 336 núcleos y una memoria GDDR5 a una velocidad de 1700 Mhz, se obtendrían mejores tiempos. Como trabajo a futuro, se plantea usar el cómputo paralelo con CUDA para calcular las características de Haar para la clasificación de objetos.

Referencias

[1] J. Fung and S. Mann, "Using multiple graphics cards as a general purpose parallel computer: Applications to computer vision," in *Proceedings of the Pattern Recognition, 17th International Conference on (ICPR'04) Volume 1 - Volume 01*, ser. ICPR '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 805–808.

[2] J. Farrugia, P. Horain, E. Guehenneux, and Y. Alusse, "Gpucv: A framework for image processing acceleration with graphics processors," in *IEEE International Conference on Multimedia and Expo (ICME2006)*. Université Lyon and GET/INT, Département EPH, July 2006, pp. 585–588.

[3] R. Shams and N. Barnes, "Speeding up mutual information computation using nvidia cuda hardware," in *Proceedings of the 9th Biennial Conference of the Australian Pattern Recognition Society on Digital Image Computing Techniques and Applications*, ser. DICTA '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 555–560. [Online]. Available: http://dl.acm.org/citation.cfm?id=1332477.1333462

[4] J. Fung and S. Mann, "Using graphics devices in reverse: Gpu-based image processing and computer vision," in *2008 IEEE International Conference on Multimedia and Expo*. IEEE, June 2008, pp. 9–12.

[5] Z. Yang, Y. Zhu, and Y. Pu, "Parallel image processing based on cuda," *2008 International Conference on Computer Science and Software Engineering*, 2008.

[6] Y. Tarabalka, T. V. Haavardsholm, I. Kåsen, and T. Skauli, "Real-time anomaly detection in hyperspectral images using multivariate normal mixture models and gpu processing." *J. Real-Time Image Processing*, vol. 4, no. 3, pp. 287–300, 2009.

[7] A. K. Jain, *Digital Image Processing*. Prentice Hall, 1989.

[8] R. Crane, *A simplified approach to image processing: classical and modern techniques*. Prentice Hall, 1997.

[9] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 2nd ed. Prentice-Hall, Inc, 2002.

[10] S. Montabone, *Beginning Digital Image Processing: Using Free Tools for Photographers*. CRC Press, 2010.

[11] A. Bovik, *The essential guide to image processing*. San Diego, California: Academic Press, 2009.

[12] W. K. Pratt, *Fundamentals of Digital Image Processing*. New York: John Wiley & Sons, 1991.

[13] P. Pacheco, *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011.

[14] A. G. López, J. Delgado, and S. Castañeda, "Metodologías de paralelización en la supercomputadora cicese2000," Departamento de Cómputo Dirección de Telemática Centro de Investigación Científica y de Educación Superior de Ensenada, Tech. Rep., Febrero 2000.

[15] R. T. Rasúa, "Algoritmos paralelos para la solución de problemas de optimización discretos aplicados a la decodificación de señales," Ph.D. dissertation, Departamento de Sistemas Informáticos y Computación. Universidad Politécnica de Valencia, Valencia, España, 2009.

[16] D. Kirk and W. Hwu, *Programming Massively Parallel Processors. A Hands-on Approach*. Morgan Kaufmann Publishers, January 2010.

[17] G. Bradski and A. Kaehler, *Learning OpenCV*, ser. Nutshell Handbook. United States of America: O´ Reilly Media, Inc., 2008.