Convolución paralela con GPU

F. J. Mesa Hidalgo, M. G. Arenas

Departamento de Arquitectura y Tecnología de Computadores.
ETSI Informática y de Telecomunicación. Universidad de Granada
Granada, España

fjmh89@icloud.com, mgarenas@ugr.es

Abstract. Este trabajo presenta un proyecto de fin de carrera realizado en la Escuela Superior de Ingeniería Informática y Telecomunicación de la Universidad de Granada donde se pone de manifiesto el potencial de la computación paralela en la GPU mediante el uso de dos tecnologías concretas: CUDA y OpenCL, tomando para ello, como ejemplo real, el algoritmo de la convolución para imágenes. Se analizaron las diversas implementaciones propuestas para cada una de las técnicas y se realizó un estudio sobre los resultados obtenidos tras la ejecución de éstos en función de las imágenes y filtros de entrada. Así mismo, se tratará el presente documento como la elaboración de un pequeño manual que pueda servir de apoyo al uso que dicho proyecto de fin de carrera tiene en el actual curso académico para la asignatura Arquitectura y Computación de Altas Prestaciones.

Keywords: Computación Paralela, gpu, cuda, opencl, convolución

Abstract. This paper presents a thesis undertaken in the ETSIIT of the University of Granada where the potential of the parallel computing on the GPU is illustrated by using two different kinds of technology: CUDA and OpenCL, taking as a real example the Convolution algorithm for images. Several proposals of algorithm will be analyzed as well as the possible explanations about the obtained outcomes based upon various input images and filters. Additionally, the present document is intended to be conceived as a manual for starters, which could potentially become a support during the current academic year for the course “Arquitectura y Computación de Altas Prestaciones”.

Keywords: Parallel programming, gpu, cuda, opencl, convolution

1 Introducción

Este artículo presenta otra vertiente de los proyectos Fin de Carrera (PFC) desarrollados dentro del departamento de Arquitectura y Tecnología de Computadores de a

adfa, p. 1, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Universidad de Granada. En este caso, se trata de un proyecto con un claro carácter investigador, pero cuya aplicación está claramente orientada a docencia, por lo que se pone de manifiesto que no siempre un proyecto investigador tiene por qué tener una aplicación al ámbito investigador, sino que se le pueden buscar otros usos.

El proyecto que se presenta está relacionado con la Computación paralela [1], tanto desde un punto de vista teórico como práctico. Este punto de vista se pone de manifiesto en las titulaciones que imparte este departamento a través de varias asignaturas que están englobadas tanto en titulaciones de Grado como en titulaciones de Ingeniería, ya en extinción.

Concretamente conceptos relacionados con la Computación Paralela se introducen dentro del Grado en Informática en asignaturas como Arquitectura de Computadores, impartida por este departamento en el cuarto semestre de la titulación. Otras asignaturas como Arquitectura y Computación de Altas Prestaciones impartida dentro de la especialidad de Ingeniería de Computadores en el sexto semestre del Grado de Informática o la asignatura Programación Paralela impartida por el departamento de Lenguajes y Sistemas de la Universidad de Granada durante el sexto semestre del Grado en Informática. Y dentro de titulaciones recién extinguidas se impartían en asignaturas como Arquitectura de Computadores I y II (quinto curso de la Ingeniería informática) o Programación Distribuida y Paralela que impartía el departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada dentro de la Ingeniería Informática (cuarto curso).

En todas estas asignaturas se abordan conceptos como los diferentes modelos de programación paralela, Paso de Mensajes, Variables Compartidas, diferentes herramientas para llevarla a cabo, como OpenMP, MPI o los diferentes niveles de paralelismo, como el paralelismo a nivel de instrucción, ILP, o a nivel funcional. Este proyecto está centrado en la utilización de una tecnología novedosa para la Computación Paralela como es CUDA [2].

CUDA es una tecnología introducida por NVIDIA para la computación de propósito general con dispositivos de procesamiento gráfico, GPU (*Graphics Processing Unit).* Este tipo de tecnología proporciona una herramienta para utilizar las GPU para acelerar programas no necesariamente gráficos, sino de ámbito general, naciendo así lo que se ha dado a llamar GPGPU *Computing (General Purpose Graphics ProcessingUnits).*

El objetivo de este proyecto era realizar un estudio de la escalabilidad que se podía obtener en una GPU utilizando diferentes tecnologías: CUDA y OpenCL [3]; realizando para ello un caso de estudio con varias tarjetas gráficas y un algoritmo determinado. Concretamente el algoritmo utilizado es una Convolución de imágenes [4]. La aplicabilidad de este proyecto ayudará a los alumnos de la asignatura Arquitectura y Computación de Altas Prestaciones a entender el un concepto importante dentro de lo que es la Computación Paralela como es la adaptación o *mapeo* de la cantidad de trabajo que un algoritmo debe realizar con la máquina o arquitectura paralela que va a llevarla a cabo.

Para ello, se detallará el proyecto realizado incluyendo detalles de las diferentes versiones de Convolución que se han desarrollado para ejecutarlas dentro de una GPU y la propuesta de práctica que se va a realizar para que los alumnos de la especialidad

de Ingeniería de Computadores lleguen a entender el concepto de *mapeo* realizando ellos nuevas versiones de los algoritmos propuestos donde la unidad o cantidad de trabajo que cada una de las hebras del algoritmo ejecuta sea variable y poder así estudiar el impacto que el *mapeo* de una tarea puede tener sobre la ejecución de un algoritmo, concepto muy relacionado con la granularidad de dicho algoritmo.

Este artículo está compuesto por 10 secciones que incluyen un pequeño estado del arte, una relación de las herramientas existentes para abordar GPGPU *Computing,* características de la arquitectura de una GPU a tener en cuenta para desarrollar aplicaciones para ellas, detalles de las diferentes versiones del algoritmo desarrollado dentro el proyecto Fin de Carrera, la practica propuesta y unas conclusiones que en cierto modo recopilan el beneficio esperado por esta iniciativa.

2 Estado del Arte

La computación acelerada por GPU se basa en el uso de una unidad de procesamiento de gráficos (GPU) junto con una CPU para mejorar significativamente el rendimiento de aplicaciones científicas [5], analíticas [6], de ingeniería [7] y empresariales [8].
En 2006 nace la primera GPU cuya arquitectura aunaba la capacidad de renderizar imágenes en 3D en tiempo real y la posibilidad de ejecutar programas escritos en C mediante el modelo de programación CUDA.

En 2007, la conocida compañía de tarjetas gráficas NVIDIA estuvo al frente en el desarrollo de las GPU que ahora potencian centros de datos con eficiencia energética en laboratorios gubernamentales [9], universidades [10], grandes compañías y pequeñas y medianas empresas de todo el mundo [11]. Las GPU están acelerando las aplicaciones en plataformas en automóviles [12], teléfonos móviles y tablets [13], aviones no tripulados y robots [14].

Las GPU han evolucionado hasta el punto de disponer de un extenso número de núcleos de ejecución para fines no solo gráficos, sino para la ejecución de programas de diversa índole [15].

A partir de ese momento surge una tendencia que, a día de hoy, sigue en auge [16]. Investigadores y desarrolladores han acogido con entusiasmo el concepto de computación en GPU para un gran rango de aplicaciones. Su buena relación coste-rendimiento ha permitido que su adopción continúe expandiendo el número de autores y universidades interesados en enseñar CUDA, como ha ocurrido en esta misma Universidad. Los propios desarrolladores contribuyen en esta tarea de expansión mediante la creación de librerías y utilidades y la cooperación vía foros de discusión [17].

3 Arquitectura GPU

A continuación, se va a realizar una presentación de las primeras nociones sobre paralelismo en la GPU con CUDA. Se presentará el modelo lógico, el modelo físico y la conexión entre ambos.

En primer lugar, se presenta el concepto de hebra o *Thread*, la cual constituye una unidad de ejecución en el dispositivo o device. Del mismo modo, se presentan los conceptos de rejilla/retícula o *Grid* y bloque o *Block*.

A nivel lógico, CUDA establece un modelo de ejecución que comprende las diversas estructuras que la *Figura 1* representa. Estas estructuras están relacionadas con la forma en la que se asignan los recursos para una determinada ejecución.

Cada función que se ejecuta en el device se denomina *kernel*. Los *kernel*s se ejecutan organizados en un Grid. Este puede estar organizado a lo largo de hasta 2 dimensiones que albergan una serie limitada de bloques. Cada bloque posee el mismo número de hebras, las cuales se distribuyen a lo largo de hasta 3 dimensiones (altura, anchura, profundidad). Cada hebra ejecuta una instancia del *kernel*, es decir, el código escrito en la función.

El número de bloques por *Grid*, así como de hebras por bloque, es limitado y depende de la generación del *device* (limitación hardware).

Además, la unidad de ejecución física se denomina *warp* y está formado por 32 hebras donde la ejecución es la misma en todas ellas.

A nivel hardware, toda tarjeta gráfica NVIDIA compatible con CUDA dispone de una serie de *Streaming Multiprocessors* o SM, donde se organizan los recursos (memoria, procesadores, contador de programa, etc). Cada SM dispone de una serie de SP que se encargan del cómputo de una hebra.

La conexión entre la parte lógica y física de CUDA se llevaba a cabo mediante la asignación de bloques al SM y la organización de los bloques en *warps*. La cantidad de bloques asignados en un momento dado dependerá de la cantidad de hebras que el SM puede ejecutar concurrentemente.

Por ejemplo, la NVIDIA GeForce GT200, es un dispositivo que aceptaba un total de 1024 hebras en ejecución. Ello condiciona la cantidad de bloques que podrán coexistir en el SM a la vez. Si el bloque contiene 16x16=256 hebras cabrán 1024/256=4 bloques que se ejecutarán al mismo tiempo. Cuando un bloque es asignado a un SM, este bloque permanecerá activo hasta que todas las hebras hayan finalizado.

Actualmente, el número de SP de los que consta cada SM suele ser 32, uno por cada hebra de un *warp*. Las ganancias que se obtienen gracias a la GPU residen, en su mayor parte, en la forma en la que se ocultan las latencias al acceder a memoria. Esto se debe a que la unidad de ejecución hardware es el *warp*. En el momento en el que alguna hebra del *warp* debe esperar una latencia por haber accedido a memoria, ese *warp* sale de ejecución. En su lugar, se elige de forma inmediata otro *warp* cuyas hebras estén listas. Esta característica conduce a una nueva situación a propiciar: maximizar el número de *warp*s por bloque. Gracias a ello, se dispone de un rango más amplio de *warps* entre los que alternar, ocultando siempre los tiempos de acceso a memoria (*latencyhiding*).

Otro factor a destacar, es el número de bloques que físicamente el dispositivo puede ejecutar o *block-slots*. De forma que si la GT200 tiene una limitación de 8 slots, un tamaño de bloque de 8x8=64 provocaría un desperdicio de los recursos. Esto es debido a que el número de hebras que la GT200 puede manejar concurrentemente es 1024. 64 hebras por bloque originaría 1024/64=16 bloques, pero ya que solo es capaz de

albergar 8, otros 8 se quedarían sin ejecutarse, desperdiciándose así 8x64=512 hebras, la mitad de las que el SM soporta.

Además, cada SM cuenta con un determinado número de registros que pueden usarse durante la ejecución de un *kernel*. Como se ha mencionado antes, cada hebra ejecuta una instancia del *kernel*, que emplea en su código un determinado número de registros locales. El programador debe velar porque el número de registros total correspondiente a las hebras que van a residir en el SM concurrentemente no supere la limitación impuesta por el hardware. Si las 1024 hebras que la GT200 puede mantener van a ser usadas y cada SM de la GT200 posee una limitación de 32786 registros, quiere decirse que cada hebra no puede usar más de 32768/1024=32 registros. Si este número es mayor habrá ciertos bloques que no podrán entrar al *device* por falta de registros, con el correspondiente desperdicio de recursos asociado.

Del mismo modo, cada *device* dispone de una cantidad limitada de memoria global, memoria constante y memoria compartida, cada una con sus propias peculiaridades (ver Figura 1). La memoria global será la memoria de propósito general que albergue los datos de un algoritmo que se esté ejecutando en la GPU. Sus ratios de transferencia son superiores a los que la memoria constante o compartida pueden aportar. Se usará memoria constante para acceder a datos de memoria con una frecuencia mucho mayor, ya que provee menores tasas de transferencia. Por último, la más veloz de los 3 tipos es la memoria compartida, que es la que usan todas las hebras de un mismo bloque.

Esta memoria compartida necesita una inicialización previa desde memoria global que tiene lugar durante la ejecución del *kernel*. La cantidad está ligada al *device*, por lo que será necesario dotar a nuestros algoritmos CUDA de cierta adaptabilidad para que puedan ser ejecutados en las diferentes tarjetas NVIDIA del mercado.

Como se ha especificado anteriormente, cuando un *warp* se ejecuta en la GPU, si el algoritmo requiere datos de entrada, éstos han de pasar por memoria global, pues no existe un acceso directo desde el device a memoria principal de la CPU durante la ejecución del *kernel*. En el caso concreto del algoritmo de la convolución, será necesario el envío de la imagen a tratar al device, organizando sus *pixels* de forma secuencial formando un array unidimensional de tamaño anchuraxalturaxprofundidad de la imagen. Será necesario establecer un convenio que permita saber en todo momento la correspondencia de cualquier pixel que se exprese de la forma Imagen[0][0][0] (matriz tridimensional) al de una matriz unidimensional.

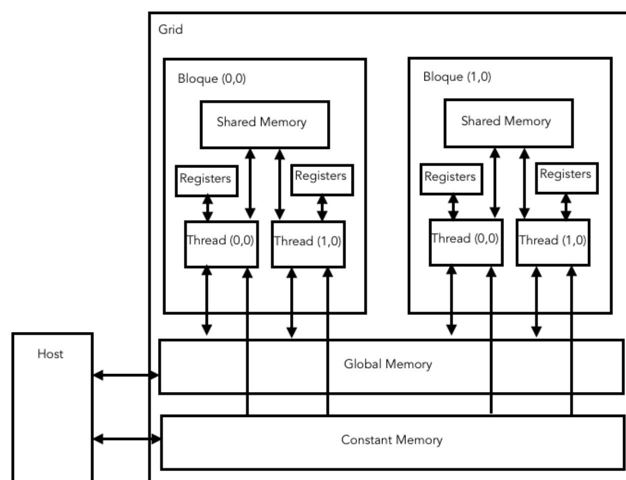

Fig. 1. Organización de la memoria en el *device*

4 Proyecto: convolución 2D en imágenes mediante la GPU

Este proyecto se centra en la computación en GPU para realizar una convolución 2D en imágenes a color.

Una convolución [4] es un operador matemático [18] que transforma dos funciones [19] f y g en una tercera función que, en cierto sentido, representa la magnitud en la que se superponen f y una versión trasladada e invertida de g. En concreto, este proyecto trata un determinado tipo de convolución: la convolución discreta bidimensional.

El proceso de convolución aplica una ventana rectangular (de un tamaño generalmente muy inferior a las dimensiones de la imagen) a la imagen, centrando la ventana en cada uno de los *pixels* de la imagen consecutivamente. Es decir, tomamos un pixel de la imagen y nos quedamos con todos los *pixels* que lo rodean. Llamaremos a este bloque $IMG_{xy}[i][j]$, donde:

- *x, y* son las coordenadas del pixel que se esta procesando en este momento.
- i, j son los índices horizontal y vertical dentro de la matriz de los elementos que rodean al pixel que estamos procesando. Así, IMGxy[1][1] es el valor del pixel actual. IMGxy[1][0] es el valor del pixel encima del actual, etc.

Además, tenemos una matriz de Coeficientes *coef[i][j]*, que para nuestros ejemplos serán matrices 3x3. Con estas definiciones, la operación a realizar para cada pixel de la imagen (es decir, para cada *y* en la dimensión vertical y cada *x* en la dimensión horizontal) será:

nuevo_pixel=IMG_{xy}[0][0]**coef*[0][0] + IMG_{xy}[0][1]**coef*[0][1] + ... + IMG_{xy}[2][2]**coef*[2][2];

Será una condición necesaria que los valores obtenidos para un pixel determinado no excedan los valores [0-255], truncando el valor si fuera necesario. En concreto, el algoritmo secuencial en nuestro caso tendría el siguiente aspecto:

```
for( ) //Rojo, Verde, Azul
for( ) //filas
for(){  //columnas
for() //filas máscara
for() //columnas máscara
if( dentro de los limites )
suma parcial
sustituir pixel por la suma
}
```

Lo que interesa comprender de este código es que está usando 3 bucles para poder iterar por todos los elementos del buffer realizando las indexaciones necesarias. Además, para cada uno de ellos se necesitan dos bucles para poder iterar sobre la máscara o *kernel*(vecindario). Como tratamiento de bordes se ha optado por simular un marco negro del mismo grosor que la máscara. Es decir, para los elementos de la máscara que no se correspondan a ningún pixel en la imagen por haber excedido los límites de ésta no se tendrán en cuenta, o lo que es lo mismo, sumaremos 0 (negro).
La convolución, combinada con la máscara adecuada, puede originar resultados muy diferentes [20]. En función de la máscara, podrá usarse para la eliminación del ruido de las imágenes o el suavizado del mismo, difuminar rostros, identificar los elementos de una imagen o enfocar imágenes desenfocadas entre otras. Es un algoritmo que interviene en numerosos proyectos cuyos resultados nos rodean a menudo (efectos fotográficos, películas, etc) y es aplicado directamente por una gran cantidad de programas informáticos; desde la suite de Adobe (*Premiere, Photoshop*) [21] o GIMP [20] hasta MATLAB [22] o *Mathematica* [23].

5 Paralelización del problema

5.1 Paralelización CUDA I: Distribución unidimensional

El enfoque a seguir consiste en que cada hebra aplique el algoritmo de la convolución a un pixel en concreto (ver Figura 2). Esto quiere decir que trabajará sobre 3 elementos de la matriz (en realidad, como se ha descrito antes, es un array, pero por simplicidad, se llamará matriz en las ocasiones que ayuden a un mejor entendimiento). Es decir, para el pixel que se corresponda a la hebra actual se aplica el algoritmo de la convolución accediendo a cada uno de los elementos de su vecindario y de los elementos de la máscara. Esta operación se repite para el resto de elementos que consti-

tuyen el pixel, es decir, para el elemento que forma la parte roja del pixel, la verde y la azul.

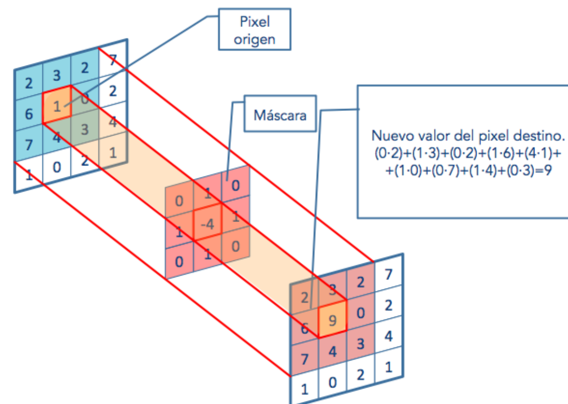

Fig. 2. Convolución 2D en imágenes.

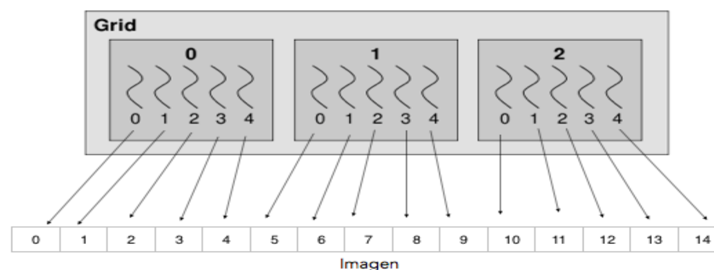

Fig. 3. Asignación de trabajo por hebra.

En una primera implementación de este algoritmo se propuso una organización unidimensional tanto de bloques como de hebras. Esta versión crea tantas hebras como *pixels* contiene la imagen. Posteriormente se crean tantos bloques como fueran necesarios para albergar las hebras tendiendo en cuenta las limitaciones:

Además, esta correspondencia entre hebra y pixel trata a cada pixel como un simple elemento de trabajo y no como un elemento cuya ubicación en la matriz está estrechamente relacionada con su vecindario. Esta asignación se puede apreciar en la Figura 3. Esto implica un mejor aprovechamiento de las hebras de cada bloque porque todos los bloques contienen hebras útiles (hebras que se corresponden a elementos de la imagen, *warps* que aprovecharán todos los SP del SM) salvo, en el peor de los casos, el último bloque si el número de *pixels* total no fuera divisible entre el tamaño de bloque decidido .

No obstante, esta primera versión desestimó el uso de memoria compartida y las condiciones de organización que ésta necesita para su uso.

5.2 Paralelización CUDA II: distribución bidimensional

El enfoque sobre el trabajo que cada hebra va a realizar se mantiene: cada hebra trabajará sobre los 3 elementos que constituyen un pixel.

La diferencia en esta ocasión estriba en que no se trata cada elemento de la imagen de forma independiente desligándolo de la semántica del problema. En cambio, en esta versión del algoritmo se aplica una disposición bidimensional a nivel de bloque y de hebra donde se podrá apreciar una correspondencia más intuitiva entre los bloques de CUDA y la imagen, donde cada uno trabaja sobre una región de la imagen de idénticas proporciones. Esta nueva organización puede apreciarse en la Figura 4.

En este enfoque existen ocasiones donde la última fila y/o columna de bloques no aprovechen todas sus hebras debido a que no haya una división exacta entre la anchura del bloque con la anchura de la imagen y/o entre la altura del bloque con la altura de la imagen. Concretamente, en este ejemplo se muestra una imagen de tamaño 5x5 donde cada bloque se compone de 4 hebras que trabajan sobre 4 *pixels*. Los últimos bloques solamente trabajan sobre 2 hebras (1 en el caso de la esquina inferior derecha) ya que no queda más imagen a tratar.

En esta ocasión, cada hebra se vale también de la componente *y* de sus coordenadas como hebra dentro del bloque y de sus coordenadas como bloque dentro del Grid, pues ahora la ubicación del pixel se realiza bidimensionalmente.

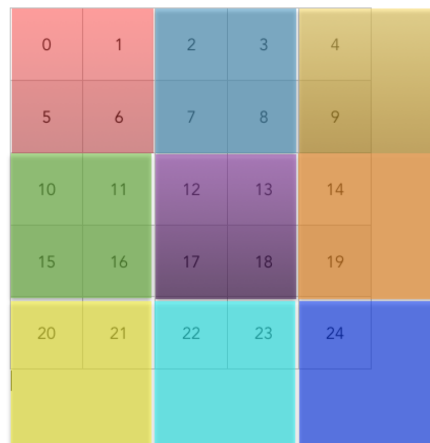

Fig. 4. En esta versión los bloques son bidimensionales y trabajan sobre áreas también bidimensionales de la imagen.

Así pues, tenemos que, para localizar el pixel que se corresponde concretamente a una hebra, en general sus filas y columnas son:

```
fila= blockIdx.y x altura_bloque  +  threadIdx.y;
columna = blockIdx.x x anchura_bloque  +  threadIdx.x;
```

Donde `altura_bloque` y `anchura_bloque` determinan las dimensiones de cada bloque. Estas dimensiones se discutirán en breve. De esta manera, tenemos que, para acceder a un pixel en concreto en el buffer secuencial, bastaría con hacerlo de forma: `buffer [fila x anchura_imagen + columna`. Teniendo en cuenta estas modificaciones, el algoritmo CUDA equivalente al secuencial sería el siguiente:

```
__global__ void convolve( . . . ) {

for( ){   //Rojo, Verde, Azul
for( )   //filas máscara
for()   //columnas máscara
suma parcial
Sustituir pixel por la suma
   sum = 0;
}
```

En resumen, el algoritmo aplica el algoritmo de la convolución para los 3 elementos (uno por cada color) de un pixel en concreto. La correspondencia hebra-pixel nos lleva siempre al primero de estos (el rojo según la representación interna que hemos seguido) y posteriormente se tratarán el verde y el azul.

Se puede apreciar cómo los dos bucles anidados que iteraban la matriz global han desaparecido, pues ahora no existe una iteración como tal sino que, en función de la hebra que esté ejecutándose, se halla una correspondencia directa con un pixel en concreto de la matriz de la imagen. Mediante la creación de un número de hebras mayor o igual al número de *pixels* nos aseguramos de que todos los *pixels* de la imagen van a ser tratados.

Respecto a la forma del bloque se ha optado siempre por respetar las proporciones cuadradas ya que es la opción que permite un mayor rendimiento a la hora de cargar *pixels* en la memoria compartida.

En cuanto al tamaño, se seguirán las mismas restricciones que las impuestas en la versión paralela anterior, pues el hecho de que el bloque sea unidimensional o bidimensional no cambia el número de hebras que lo compone y, por tanto, su repercusión en el SM.

Como se ha mencionado en los fundamentos teóricos de CUDA y OpenCL, la cantidad de memoria, ya sea global, compartida o constante, depende del dispositivo sobre el que estamos trabajando, de modo que se trata de un factor que requiere de una consulta en tiempo de ejecución. En problemas que requieren trabajar sobre unos datos de entrada, como en el caso de la convolución, la cantidad de memoria global libre determinará el tamaño máximo del problema, pues será en ella donde se aloje la imagen de entrada y de salida. Además, es un parámetro que puede variar a lo largo de una misma ejecución, pues, concurrentemente, nuestro ordenador puede estar ejecutando varios procesos que requieran también de la GPU. En nuestro caso, son 3 las reservas necesarias para albergar, respectivamente, la imagen principal, la imagen de salida y la máscara.

La memoria constante es una memoria de reducido tamaño (normalmente 64 KB), de solo lectura que proporciona un ancho de banda mayor que el de la memoria global. Es por esto y por el elevado número de accesos por hebra que es la candidata perfecta para hospedar la máscara, pues ésta no excede los 64. Esta memoria se deberá declarar como una variable global, fuera de cualquier ámbito, y será inicializada antes de la llamada al *kernel*.

Respecto a la memoria global, queda por discutir las reservas de la imagen de entrada y la imagen de salida. Como se ha establecido, la cantidad de memoria disponible es un factor que limita el tamaño de nuestro problema. Ya que si usamos la computación GPU es, precisamente, para obtener ganancias considerables con problemas de gran tamaño, no podemos permitir que la cantidad de memoria libre nos limite el rendimiento en este tipo de problemas. Se crea un algoritmo que, efectivamente, compruebe la cantidad de espacio libre del que disponemos. En función de éste, la imagen se divide (presumiblemente por filas) en piezas que puedan caber en el device. El procesamiento se lleva a cabo mediante sucesivas llamadas al *kernel* que procesan cada una de las partes de la imagen independientemente. Cada resultado se incorporaría a una matriz global donde se irían acoplando el resto de las partes. Este hecho plantea un problema que no existe en el algoritmo secuencial respecto al tratamiento de bordes. Ya que el algoritmo secuencial trabaja sobre una misma imagen sin dividir debido a que ésta se encuentra en memoria principal, los *pixels* que necesitaban de un vecindario que se escapaba de los límites de la imagen no formaban parte de la suma (es decir, se sumaba 0, el color negro). En esta versión del algoritmo, si vamos a dividir la imagen en piezas tenemos que tener en cuenta que, si lo hacemos en filas, los bordes horizontales no pueden ser tratados de la misma manera. Es decir, no podemos fingir la presencia de un marco negro cuando en realidad ese marco son *pixels* de la parte anterior y/o siguiente de la imagen.

Las imágenes de mayor dimensión no serán aptas para todos los dispositivos ya que no podrán almacenarse en su totalidad en éste. En estos casos, la imagen que será procesada en la GPU será una de las particiones de la imagen original a la que se le han añadido los bordes necesarios para llevar a cabo la convolución de forma coherente.

5.3 Paralelización CUDA III: distribución bidimensional

La motivación por la implementación de un algoritmo que use memoria compartida reside en los resultados que podemos obtener debido a las bajas latencias que ésta posee respecto a memoria global. Sin más preámbulos, se procede a mostrar la propuesta de algoritmo:

```
__global__ void convolve ( ** Md imagen . . .  ) {
      extern __shared__ unsigned char Mds[];
      for( ){   //Rojo, Verde, Azul
      //Loading the pixels
      Mds[pixelLocal] = Md[pixelActual];
      <cargar los pixels de los bordes>
```

```
__syncthreads();
for( )   //filas máscara
for( )   //columnas máscara
suma parcial accediendo a Mds
Sustituir pixel por la suma
__syncthreads();
}
```

En él se resaltan en rojo las modificaciones aplicadas al algoritmo CUDA anterior:

- Primeramente se declara el buffer compartido Md. La palabra reservada "*extern*" permitirá asignarle un tamaño dinámicamente en función de la limitación de memoria compartida del device. Este parámetro se asignará en el momento de invocar al *warp*.
- Posteriormente se procede al trasladado del pixel desde la imagen en memoria global a memoria compartida.
- A continuación, si la hebra actual representa un pixel de los bordes se traerían los bordes implicados a memoria compartida.
- Tras ejecutar estos puntos se establece una barrera de sincronización, ya que hasta que todos los *pixels* estén cargados no se procederá a su cómputo, pues hasta ese momento la memoria contendrá datos sin especificar.
- Tras su cómputo es necesario establecer otra barrera de sincronización. De lo contrario, las hebras que hayan finalizado este cómputo comenzarán a cargar el elemento del siguiente color, perjudicando cómputo del color anterior de sus hebras vecinas.

6 Experimentos

Se han elegido 4 tamaños de imagen diferentes, siendo éstas de proporciones cuadradas. Sus tamaños son, en orden ascendente: 2800x2800px, 5600x5600px, 11200x11200px y 22400x22400px.

Además, se ha contado con 3 tipos de máscara diferentes que apliquen el algoritmo de la convolución de tamaños 3x3, 5x5 y 7x7.

Cada una de las posibles combinaciones entre imagen y filtro van a ser ejecutadas en las diferentes versiones del algoritmo: algoritmo secuencial, algoritmo CUDA, algoritmo CUDA que usa memoria compartida y un algoritmo en OpenCL que no se ha detallado en este artículo pero sí se utiliza para comparar.

En cuanto al hardware empleado, se trata de un PC con la distribución Ubuntu 12.04 (Linux 3.5.0-36-generic) cuyas características CPU más trascendentes son [24]:

- Nombre de modelo: Intel® Core™ 2 Quad CPU Q6600
- Número de procesadores: 1
- Núcleos CPU: 4
- CPU MHz: 2400

En cuanto a las características de la GPU, podemos destacar:

- Nombre: GeForce GTX 660Ti
- Generación: 3.0
- Nº máximo de hebras por SM: 2048
- Nº de SM: 7
- Memoria global: 2GB
- Nº máximo de registros: 65536

A diferencia de CUDA, OpenCL no nos proporciona mecanismos para conocer detalles concretos sobre la implementación hardware de la GPU que permitan una asignación óptima de los elementos y grupos de trabajo. Por ejemplo, a través de OpenCL no podemos saber el número de elementos de trabajo que cada SM puede permitir simultáneamente, lo cual es un factor determinante para definir la composición de los grupos de trabajo y estimar cuántos de éstos podrán caber en la máquina. Es por esto que el tamaño del grupo de trabajo elegido en OpenCL será el tamaño de bloque optimizado por el algoritmo CUDA, pues el hardware sigue siendo el mismo. Se deduce que en otros dispositivos (Intel, AMD, etc) no podremos valernos de la API de NVIDIA para optimizar el bloque, por lo que el programador deberá encontrar empíricamente el tamaño de grupo de trabajo apropiado para algoritmos OpenCL.

La medición de los tiempos se llevó a cabo mediante la ejecución de dos scripts escritos en *tcsh*, donde cada ejecución se lanza 5 veces para poder computar un promedio que normalice las ejecuciones críticas.

7 Resultados

Los tiempos obtenidos, quedan reflejados en la Tabla 1, donde la primera columna indica el tamaño del filtro aplicado y la segunda expresa el ancho y alto de la imagen.

Table 1. Tiempo medio (ms) de las 5 ejecuciones lanzadas por cada versión del algoritmo, tamaño de máscara y tamaño de la imagen.

		Sec.	CUDA (II)	CUDA (III)	OpenCL	hebras	Tamaño bloque
	2800	6194	2,13	8,83	2,14	7840000	16x16
	5600	25182	9,03	55,50	9,15	31360000	32x32
	11200	102996	36,06	202,54	30,84	125440000	32x32
	22400	414130	144,33	776,14	144,84	265932800	32x32
	2800	16524	5,25	15,23	4,27	7840000	16x16
	5600	67248	21,13	93,91	18,06	31360000	32x32
	11200	274914	84,51	332,51	60,89	125440000	32x32
	22400	1339042	300,16	1301,62	274,84	265932800	32x32
	2800	32936	9,93	25,43	7,30	7840000	16x16
	5600	169982	39,13	143,13	25,91	31360000	32x32
	11200	745736	146,37	537,32	110,38	125440000	32x32
	22400	3211858	553,62	1435,05	406,02	265574400	16x16

Table 2. Desviaciones estándar de cada uno de los tiempos.

		Sec.	CUDA (II)	CUDA (III)	OpenCL	hebras	Tamaño bloque
3x3	2800	26,08	0,01	0,01	0,00	7840000	16x16
	5600	4,47	0,01	0,01	0,00	31360000	32x32
	1200	16,73	0,00	16,00	0,00	125440000	32x32
	22400	113,36	0,01	27,51	3,66	265932800	32x32
5x5	2800	5,48	0,01	0,01	0,00	7840000	16x16
	5600	8,37	0,01	0,01	0,00	31360000	32x32
	11200	53,67	0,01	19,51	0,00	125440000	32x32
	22400	267670,62	14,72	24,02	18,39	265932800	32x32
7x7	2800	1740,99	0,01	0,02	0,00	7840000	16x16
	5600	34495,12	0,01	7,84	0,00	31360000	32x32
	11200	118934,51	13,39	4,76	7,89	125440000	32x32
	22400	1558,24	23,38	22,34	18,64	265574400	16x16

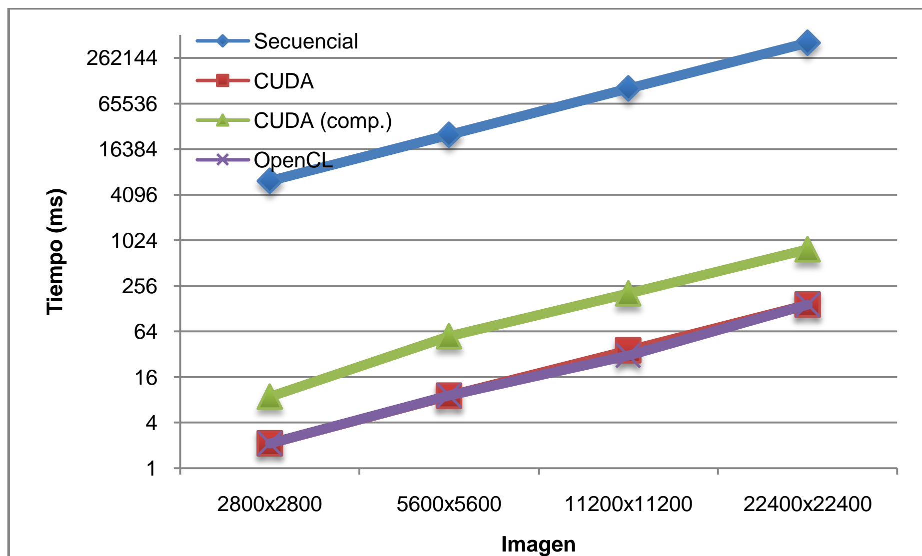

Fig. 5. Tiempos para diversas imágenes con una máscara de 3x3. (Escala logarítmica base 2.)

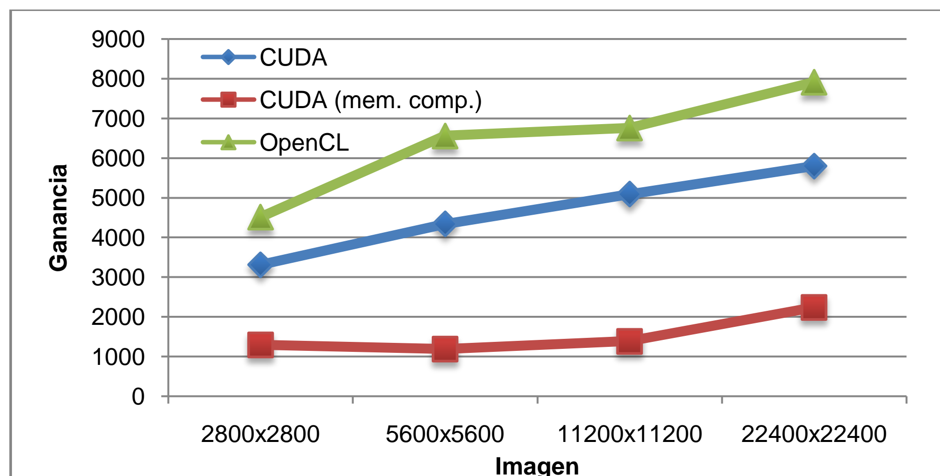

Fig. 6. Ganancias de los distintos algoritmos paralelos respecto al secuencial. Se ha tomado el filtro de dimensiones 7x7.

En la Figura 5 se puede apreciar los diferentes tiempos de ejecución obtenidos para las diferentes versiones del algoritmo. Puede observarse que la escala es logarítmica, por lo que los tiempos se mejoran de forma exponencial conforme aumenta el tamaño de la imagen a tratar. También queda de manifiesto que los tiempos obtenidos con la versión de CUDA que utiliza memoria compartida no es todo lo buena que se esperaba, resultados que se explican debido al tipo de dato utilizado en cada pixel de la imagen, puesto que se ha representado la imagen utilizando bytes para maximizar la capacidad de la GPU y esto provoca colisiones en el acceso a la memoria del *device*. Por otra parte, el coste en tiempo que supone traer los datos desde memoria global hasta memoria local no se compensa con las operaciones que se realizan con cada dato, por lo que los tiempos de la versión de memoria compartida son peores. Respecto a los tiempos de la versión en OpenCL y la versión CUDA son prácticamente similares, por lo que ambas tecnologías son capaces de proporcionar un rendimiento óptimo de la GPU.

Utilizando estos tiempos (Figura 5) hemos calculado la ganancia obtenida para cada tipo de imagen y cada versión del algoritmo (ver Figura 6). Como se puede apreciar las ganancias obtenidas son superlineales para todos los casos.

8 Práctica propuesta. Beneficios esperados.

Durante el desarrollo del PFC se tuvo en mente siempre que era un trabajo de investigación para ver hasta donde se podía explotar una GPU para realizar una tarea de tratamiento de imágenes altamente parametrizable y paralelizable. Sin embargo, a la terminación del mismo, se pensó que quizás todo ese trabajo debería ser aprovechado por otros alumnos, añadiéndole así un grado más de utilidad al proyecto para que éste pudiera ser reutilizado por otros estudiantes.

Por otra parte, los alumnos no siempre se sienten implicados o motivados a realizar PFC o Proyectos Fin de Grado, PFG, relacionados con la Computación Paralela, puesto que en la mayoría de los casos les atraen más otros proyectos o simplemente les parece una tarea de alta complejidad el abordar un trabajo de programación y optimización de código paralelo. Sin embargo creemos que plantearles una práctica que puedan realizar utilizando un código desarrollado en un PFC anterior, podría ser algo positivo que dota a la práctica de un plus para la motivación del alumnado puesto que podrán ver que compañeros como ellos son capaces de realizar un código optimizado y adaptado a una máquina tan bueno como cualquier otro desarrollado por profesores, editores de bibliografía o investigadores especializados en esta tarea y que, además, ellos son capaces de modificar, optimizar y utilizar sin problemas.

Por último, esta práctica está planteada para que manejen y apliquen conceptos implicados en Computación Paralela y que afectan claramente a la escalabilidad, eficiencia y rendimiento del algoritmo y que no en todos los casos son capaces de asimilar. Concretamente nos referimos a los conceptos de granularidad y mapeo de una tarea que debe llevar a cabo un algoritmo en una máquina concreta de ejecución.

El PFC explicado en la sección anterior está planteado asumiendo que el tamaño de una tarea básica es el procesamiento de un pixel de la imagen, es decir, cada hebra del algoritmo procesa la convolución de un pixel produciendo así un algoritmo de granularidad fina y este principio se ha mantenido en todas las versiones del algoritmo. Es decir, existirán tantas tareas independientes como *pixels* tenga la imagen a procesar y cada una de estas tareas será ejecutada por una hebra diferente. Pero, qué pasaría si nuestra tarea básica no fuera el procesamiento de un pixel, sino que este valor fuera algo variable del algoritmo. Este segundo enfoque puede apreciarse en la imagen que representa la Figura 7.

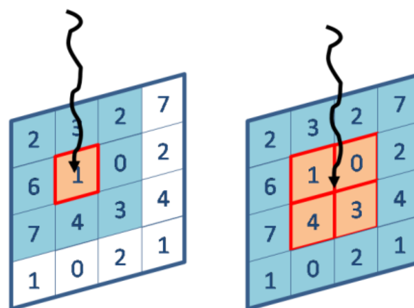

Fig. 7. En esta figura se aprecia la diferencia entre considerar una tarea básica el procesamiento de un sólo pixel, parte izquierda de la figura, frente a considerar que una tarea básica está formada por el procesamiento de 4 *pixels*, representada en la parte derecha de la imagen. En ambos casos, la tarea básica se procesa por una sola hebra de procesamiento, representado en la figura por una flecha vertical ondulada.

La práctica consiste en que el alumno sea capaz, partiendo ya de algunas de las versiones del algoritmo, modificar la granularidad del mismo, viendo cómo al cambiar la granularidad, es posible cambiar los tiempos de ejecución, tanto en un sentido, disminuyéndolos, como en otro, aumentándolos. Cuando el tamaño de la tarea básica aumenta, lo normal es que los tiempos de ejecución disminuyan, puesto que cada hebra

creada tendrá más trabajo a procesar y por lo tanto el tiempo que se ha empleado en su creación, será compensado por la cantidad de trabajo que es capaz de llevar a cabo en paralelo con las demás hebras. Sin embargo, esta disminución del tiempo compensará sólo en el caso de que el trabajo realizado por todas las hebras en paralelo no provoque otro tipo de sobrecarga adicional que ocasiona otros retrasos. Estos otros retrasos surgen, por ejemplo, cuando varias hebras necesitan acceder al mismo conjunto de *pixels* o cuando el número de hebras necesarias para procesar la imagen disminuye de tal manera, al dividirse el trabajo a realizar entre ellas, que la GPU no puede ser aprovechada de forma completa y simultánea.

El trabajo pedido a los alumnos consistirá en la parametrización del tamaño de la tarea básica y en el estudio de la influencia de este parámetro en una de las versiones del algoritmo, concretamente en la versión de CUDA que utiliza memoria global.

Los resultados esperados con esta práctica se pueden ver de manifiesto en el siguiente ejemplo que se ha llevado a cabo utilizando una imagen de 3000x4000 *pixels* en formato RGB junto con una máscara pequeña de tamaño 3x3. Los resultados se pueden ver en la figura 11, donde se puede apreciar que al ir modificando el tamaño de la cantidad de trabajo que cada hebra debe realizar se obtiene un tiempo medio de unos 183 ms, conforme aumentamos la granularidad, el tiempo de ejecución disminuye hasta valores medios de 179 ms, unos 4 milisegundos menos en media, optimización que teniendo en cuenta que la unidad de medida son los milisegundos, es apreciable. A partir de este valor, al aumentar ya el tamaño de las tareas y pasar a tamaños de granularidad superiores a 4, es decir que cada hebra procese un cuadrado de la imagen de tamaño 5x5 o 6x6 los tiempos de ejecución vuelven a empezar a subir. Es en este punto donde podemos afirmar que ya un aumento de la granularidad no compensa en términos de tiempo de ejecución. La gráfica que representa dichos resultados se puede apreciar en la gráfica siguiente.

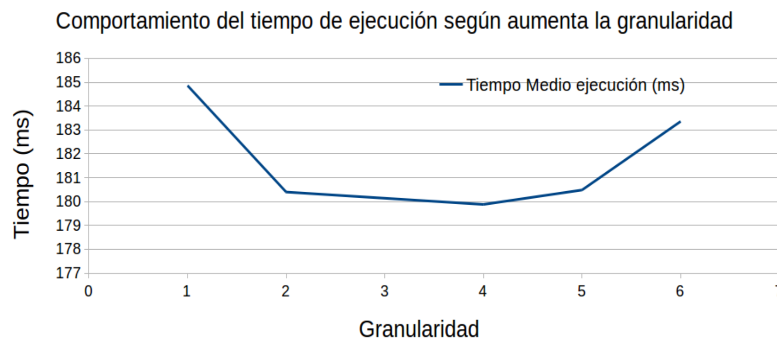

Fig. 8. Ilustración de cómo el tiempo de ejecución disminuye al aumentar la granularidad de 1 a 4. A partir de este punto, el aumentar la granularidad no mejora el tiempo de ejecución, sino que lo empieza a empeorar, puesto que el número de hebras es menor y seguramente ya no son suficientes para ocultar la latencia de los accesos a memoria.

9 Conclusiones

En el estudio realizado se puede apreciar como una GPU es capaz de alcanzar ganancia superlineales para el procesamiento de cualquier imagen, sea del tamaño que sea, teniendo en cuenta que se trata de un algoritmo muy paralelizable y que se puede adaptar fácilmente a la arquitectura paralela de una GPU. Concretamente, los resultados incluyen ganancias de más de 7000x en imágenes de gran tamaño para la versión realizada con OpenCL y de más de 7000x en el caso de la versión realizada con CUDA.

Esto pone de manifiesto que no hay que realizar ninguna presunción sobre la paralelización de algoritmos en arquitecturas tan específicas como una GPU y que cualquier límite puede ser superado sin problemas por un algoritmo bien organizado y bien estructurado adecuándolo de la mejor forma a la arquitectura que subyace en el plano de la ejecución.

Por otra parte, se plantea con este proyecto su reutilización para una de las asignaturas de la especialidad de Ingeniería de Computadores, concretamente para la asignatura "Arquitectura y Computación de Altas Prestaciones". En dicha práctica se plantea que los alumnos cambien la granularidad de una de las versiones del algoritmo paralelo que se describe en este trabajo para que realicen un estudio de la influencia de esta característica en los tiempos de ejecución. De este modo se propone una forma de abordar, por una parte, el entendimiento del concepto de granularidad y, por otra, la puesta en práctica de técnicas de programación paralela impartidas en la parte teórica de la asignatura como es la agrupación de tareas de igual tipo y tamaño, disminuyendo así el número total de tareas a realizar por la GPU y por lo tanto del número de hebras necesarias para la conclusión del trabajo de procesamiento de una imagen.

Este trabajo persigue a su vez, una reutilización de un PFC, que en la mayoría de los casos, es trabajo que no vuelve a utilizarse una vez superada dicha materia y que por lo general, el alumno deja de mantener y mejorar para dedicarse al mundo laboral o simplemente porque no se ha planteado en el diseño del proyecto que pudiera llegar a ser un producto útil, cosa que sólo ocurre en unos pocos casos de todos los proyectos fin de carrera desarrollados.

Referencias

[1] Yan Yong and Zhang Xiaodong, "Profit-effective parallel computing," *Concurrency*, vol. 7, no. 2, April 1999.

[2] NVIDIA Corporation. (2015, January) nvidia.com. *URL* http://www.nvidia.com/object/what-is-gpu-computing.html

[3] Khronos Group. (2014, December) khronos.org. *URL* https://www.khronos.org/opencl/

[4] Wikipedia Inc. (2014, September) wikipedia.org. *URL*

http://es.wikipedia.org/wiki/Convolución

[5] Javier Delgado, Joao Gazolla, Esteban Clua, and S. Masoud Sadjadi, "A Case Study on Porting Scientific Applications to GPU/CUDA," Miami, U.S.A., 2010.

[6] Ren Wu, Bin Zhang, and Hsu Meichun, "GPU-Accelerated Large Scale Analytics," HP Laboratories, 2009.

[7] Nesrin Aydin Atasoy, Baha Sen, and Burhan Selcuk, "Using gauss - Jordan elimination method with CUDA for linear circuit equation systems," Karabuk University, Engineering Faculty, Karabuk,TURKEY, 2012.

[8] Douglas C.C and Hyoseop Lee, "Basket Option Pricing Using GP-GPU Hardware Acceleration," Univ. of Wyoming, Laramie, WY, USA, 2010.

[9] R. Ammendola et al., "The GAP project - GPU for realtime applications in high energy physics and medical imaging," in *Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), 2013 IEEE*, Seoul, 2013.

[10] G. Poli, J.H. Saito, J.F. Mari., and M.R. Zorzan, "Processing Neocognitron of Face Recognition on High Performance Environment Based on GPU with CUDA Architecture," in *Computer Architecture and High Performance Computing, 2008. SBAC-PAD '08. 20th International Symposium on*, Campo Grande, MS, 2008.

[11] Inc Tabor Communications. HPC wire. *URL* http://www.hpcwire.com/2011/12/15/emerging_companies_ride_wave_of_gpu_computing/

[12] Andreas Reich and Jen-Hsun Huang. Youtube. *URL* http://youtu.be/wNSWWOf6-Hw

[13] Migel Bordallo López, Henri Nykanen, Jari Hannuksela, Olli Silvén, and Markku Vehvi ̈ainen, "Accelerating image recognition on mobile devices using GPGPU," University of Oulu, Oulu, Oulu, Finland, 2011.

[14] O. Cetin and G. Yilmaz, "GPGPU accelerated real-time potential field based formation control for Unmanned Aerial Vehicles," in *Unmanned Aircraft Systems (ICUAS), 2014 International Conference on*, Orlando, FL, 2014.

[15] NVIDIA Corporation. (2014, December) NVIDIA. *URL* http://www.nvidia.es/object/corporate-timeline-es.html

[16] John Nickolls and William J. Dally, "THE GPU COMPUTING ERA," NVIDIA, Santa Clara, CA, 2010.

[17] Stack Exchange inc. Stack Overflow. *URL* http://stackoverflow.com/search?q=CUDA

[18] Wikipedia Inc. (2014, November) wikipedia.org. *URL* http://es.wikipedia.org/wiki/Operador

[19] Wikipedia Inc. (2014, November) wikipedia.org. *URL* http://es.wikipedia.org/wiki/Función_matemática

[20] GIMP. (2013) docs.gimp.org. *URL* http://docs.gimp.org/en/plug-in-convmatrix.html

[21] Adobe Systems Incorporated. (2014, December) http://helpx.adobe.com. *URL* http://helpx.adobe.com/premiere-pro/using/video-effects-transitions.html

[22] The Mathworks Inc. (2014, December) es.mathworks.com/. *URL* http://es.mathworks.com/help/matlab/math/convolution.html

[23] Wolfram. (2014, December) reference.wolfram.com/. *URL* https://reference.wolfram.com/language/ref/Convolve.html

[24] Stack Exchange Inc. (2011, January) unix.stackexchange.com. *URL* http://unix.stackexchange.com/questions/6345/how-can-i-get-distribution-name-and-version-number-in-a-simple-shell-script