PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

FACULTAD DE CIENCIAS E INGENIERÍA

PONTIFICIA
UNIVERSIDAD
CATÓLICA
DEL PERÚ

Diseño e Implementación del Algoritmo de Convolución Bidimensional en la Arquitectura CUDA

Tesis para optar el Título de Ingeniero Electrónico, que presenta el bachiller:

Ivo Sánchez Checa Crosato

ASESOR: Paul Rodriguez

Lima, Agosto del 2011

Introducción

En el presente documento se explicarán las consideraciones realizadas para implementar la convolución bidimensional en la arquitectura CUDA. En general se discutirá la metodología seguida y se mostrarán y analizarán los resultados obtenidos.

Inicialmente en el Capítulo 1, a manera de introducción, se discutirá la programación en paralelo y los diferentes aspectos a tener en cuenta al desarrollar programas para arquitecturas concurrentes. De esta forma se pretende explicar conceptos importantes que servirán para poner la presente investigación en contexto y comprender mejor los siguientes capítulos.

En el Capítulo 2 se describirá a profundidad los aspectos más importantes de la arquitectura CUDA así como la operación de convolución bidimensional. De esta manera se espera dejar claros los conceptos pertinentes.

Posteriormente en el Capítulo 3 se explicará la metodología para el desarrollo de los programas realizados, detallándose las diferentes consideraciones para optimizar el desempeño y reducir el tiempo de ejecución de los mismos.

Finalmente en el capítulo 4 se mostrarán los tiempos de ejecución obtenidos con los diferentes programas desarrollados. Estos se obtendrán al tomar en cuenta cada una de las optimizaciones mencionadas en el tercer capítulo con lo que se apreciará la mejora de desempeño en cada caso. A continuación se tomará la mejor de las implementaciones realizadas y se comparará con otras existentes para poner los resultados obtenidos en contexto.

Por último se presentarán las conclusiones y recomendaciones pertinentes.

Capítulo 1
Estado del Arte

1.1. Marco Teórico

Las computadoras modernas poseen un gran poder de cómputo comparadas con las de hace pocos años. Resulta común encontrar arquitecturas multiprocesador de 2, 4 y hasta 8 núcleos con velocidades de reloj que sobrepasan los 3GHz en computadoras personales [10]. No obstante, existe aún un gran número de aplicaciones con performance bastante pobre debido a la gran cantidad de cálculos y operaciones involucradas. Considérese por ejemplo la codificación de video [10], las representaciones gráficas de grandes cantidades de datos y las simulaciones de procesos biológicos entre otros [5].

La necesidad actual por programas mas rápidos, aplicaciones con miles de usuarios en simultáneo y juegos y películas cada vez más semejantes a la realidad, sólo pone en evidencia que es necesario que la capacidad computacional siga aumentando para satisfacer las necesidades de los usuarios. Más allá de las aplicaciones para el público en general, cabe mencionar el hecho que muchas personas utilizan sus PCs para realizar simulaciones con grandes cantidades de variables, las cuales requieren una alta capacidad de procesamiento.

En este contexto, los fabricantes de procesadores como Intel, AMD, PowerPC y Sparc han dejado de lado sus métodos tradicionales de mejorar la capacidad computacional de sus microprocesadores aumentando la frecuencia de trabajo de los mismos. Esto se debe en parte a limitaciones de consumo de energía, de espacio y de disipación de calor [5]. En contraposición han empezado a centrar sus esfuerzos en arquitecturas multiprocesador que puedan ejecutar varias instrucciones en simultáneo [11]. De esta forma, los usuarios comunes están

empezando a tener acceso a características con las que previamente sólo contaban las supercomputadoras. Esto es, procesadores multinúcleo o multiprocesadores.

Estos cambios han dado lugar no sólo a cambios en la manera en que se diseña el hardware, sino que han producido un cambio de paradigma en la manera en que los programadores desarrollan software [12]. Esto se refleja en el hecho que estos ya no pueden contar con que sus aplicaciones serán más rápidas con cada nueva generación de procesadores [11].

Conocer y hacer uso de los niveles de paralelismo del hardware en que se desarrolla se torna imperativo si se pretende llevar el desempeño de las aplicaciones a nuevos límites. Se dice incluso que en el futuro, la programación en paralelo será la norma [10]. Por esta razón, se evidencia la importancia del estudio de técnicas de programación en paralelo, sus métodos y sus posibles desventajas y dificultades. Las siguientes secciones brindan una idea breve de estos tópicos.

1.1.1. Conceptos

En sistemas operativos modernos, cuando se ejecuta un programa, se carga el mismo o parte de este en un espacio de memoria previamente separado. A esto se le conoce como proceso, el cual se puede definir como un programa en ejecución [10]. En realidad, además del programa en sí (esto es sus instrucciones y datos), un proceso está compuesto por los registros que utiliza el mismo y un contador de programa (PC por sus siglas en inglés) que apunta a la posición de memoria que contiene la siguiente instrucción a ejecutar.

Un proceso puede subdividirse en *threads* que son flujos de control dentro de un mismo proceso [10]. Es decir, un proceso consta de al menos un *thread*.

Estos *threads* pueden asignarse de manera independiente a múltiples recursos de procesamiento, haciendo factible la ejecución de varios de manera simultánea. Otra particularidad, es que los *threads* pueden compartir espacios de memoria, es decir, determinado *thread* puede escribir un dato en memoria compartida y otro podrá leerlo en seguida.

Finalmente, por su naturaleza de ser una subdivisión de un proceso, un *thread* es normalmente más ligero que este último [10]. Con esto se quiere decir que, si bien un *thread* tiene también variables locales e información propia, esta es menor que la que puede llegar a tener un proceso.

Tradicionalmente, la creación y manejo tanto de *threads* como de procesos supone una demora debido a los ciclos de reloj en los que la(s) unidad(es) de procesamiento están ocupadas en tareas de manejo de memoria, recursos y coordinación de los mismos. Esta carga extra debido a manejo de unidades de ejecución se denomina *overhead* y es proporcional a la cantidad de *threads* con los que se trabaja. Por esta razón el programador debe encontrar un número óptimo para alcanzar un equilibrio entre el *overhead* generado y el nivel de paralelización de la tarea [10].

1.1.2. Computación en paralelo

En general la computación concurrente pretende ejecutar tantas porciones de un proceso como sea posible de manera simultánea. Esto depende, por supuesto, del tipo de aplicación o algoritmo que se esté implementando, por lo que es difícil presentar una receta para desarrollar software concurrente que se aplique en todos los casos.

Es factible, sin embargo, definir modelos para catalogar distintos tipos de problemas y estudiarlos de manera eficaz. En [12] se separan los algoritmos en

tres grupos según su potencial paralelismo. Se tienen pues, las tareas que presentan paralelismo independiente, las que presentan un paralelismo estructurado y las que en contraposición a estas últimas presentan un paralelismo no estructurado.

El primer tipo se presenta en aplicaciones cuyo paralelismo puede resultar evidente a simple vista. Se trata de procesos que se pueden dividir en tantos subprocesos como elementos haya que procesar sin afectar el resultado final de la operación. Para un ejemplo cotidiano, considérese un grupo de estudiantes que debe repartir 500 volantes en un campus universitario. Suponiendo que entregar un volante toma un determinado tiempo t, un estudiante entregará todos los volantes en un tiempo igual a 500t. Si esta persona pide ayuda a 4 compañeros, podrán hacer el mismo trabajo en un tiempo de 100t. Así, mientras mayor sea la cantidad de estudiante colaborando con esta tarea (con un límite de 500), menor será el tiempo que tome la misma. La idea a no perder de vista en este ejemplo es el hecho que el proceso de repartir un volante específico es independiente de la entrega de los otros, de ahí que este tipo de paralelismo se denomine independiente. Como se verá en el siguiente capítulo, el algoritmo de convolución en dos dimensiones en el que se centra este documento presenta un paralelismo de este tipo.

El siguiente tipo de paralelismo es el estructurado. En este caso, las operaciones a ejecutar en un elemento específico dependen de otros elementos y viceversa. Por esta razón la coordinación entre las distintas unidades de procesamiento es vital para asegurar que no se modifique ningún elemento sin antes haber realizado todas las operaciones que dependen del mismo. Para evitar esto último, se requiere cierto grado de sincronización entre los subprocesos.

Por último, el paralelismo no estructurado se presenta en aplicaciones que pueden dividirse en múltiples subprocesos, pero en las que cada uno de estos

ejecuta tareas distintas. Aquí el nivel de sincronización debe ser mayor que en el caso de paralelismo estructurado, pues será necesario especificar en qué momento un subproceso deberá esperar el resultado de otro o “bloquear” el acceso a un determinado elemento mientras uno de los subprocesos lo utiliza.

1.2.3. Alternativas a CUDA

a) OpenMP

OpenMP [16] es un API cuyas características pretenden facilitar el desarrollo de programas paralelos. Consta de directivas de compilador, librerías en tiempo de ejecución y variables de entorno que facilitan la descripción de una o más porciones de un programa que deben ser ejecutadas por varias unidades de procesamiento de manera simultánea. Asimismo permiten especificar los recursos a utilizar (tamaño de memoria compartida, cantidad de hilos o threads de ejecución) en cada una de estas unidades y determinar de qué manera se dividirá la carga de trabajo entre las mismas. Lejos de ser un lenguaje de programación provee a los desarrolladores de una notación especial que puede ser utilizada en programas escritos en C, C++ o FORTRAN [1].

Entre sus principales ventajas se encuentra el hecho de ser multiplataforma pues soporta una variedad de sistemas operativos (UNIX, Linux, Solaris, Windows) así como un alto número de compiladores, entre los que cabe hacer referencia a GNU gcc, xl de IBM y los compiladores de las herramientas de desarrollo Solaris Studio de Oracle y Visual Studio C++ de Microsoft [16].

Por otro lado, en su nivel más simple, puede utilizarse OpenMP para paralelizar programas secuenciales sin grandes cambios al código fuente del mismo más allá de un pragma o directiva de compilador para señalar el inicio y/o fin de una sección de código a paralelizar. Para esto basta con agregar la directiva

#pragma omp antes del bucle a paralelizar y el compilador se encargará de paralelizarlo. Por supuesto estas instrucciones cuentan con un rango de parámetros y opciones que escapan de los alcances de este documento.

Una primera desventaja de OpenMP es el alto costo computacional (*overhead*) "inaceptable" presente al paralelizar aplicaciones con cantidades "relativamente grandes" de *threads* [1]. Si bien el texto citado no especifica un número de *threads* considerado alto, se nota ya una clara diferencia con el modelo de programación de CUDA, en cuyo modelo de desarrollo prima el concepto de dividir la carga de trabajo en la mayor cantidad de unidades de procesamiento posibles. Otra desventaja que descalifica a OpenMP en lo que respecta a los alcances de la presente tesis es el hecho que no soporta GPUs directamente.

b) OpenCL

Tal vez la alternativa más clara a CUDA es OpenCL [17] del grupo Khronos. Este es un framework para programación paralela cuya principal ventaja es que se puede usar no sólo para programar GPUs de distintos fabricantes (incluyendo NVIDIA) sino también CPUs y otros dispositivos (Procesadores DSP y FPGAs).

A grandes rasgos, el modelo de programación de OpenCL es bastante similar al de CUDA. Se considera incluso una "correspondencia de uno a uno" entre ambos [5], pues su manera de organizar la jerarquía de memorias es casi idéntica.

La idea principal de OpenCL es ser portable, es decir permitir al desarrollador trabajar en la mayor cantidad de arquitecturas posibles. El costo de esta ventaja es, sin embargo, mayor complejidad para el programador, principalmente porque se deben manejar varios dispositivos, muchas veces heterogéneos.

Esto impacta también en la *performance* de las implementaciones desarrolladas, pues muchas de las directivas u opciones que le permiten a estas alcanzar altos desempeños no están disponibles en todos los dispositivos. Es decir, para alcanzar buenas prestaciones de manera portable, se necesita código que determine las características del dispositivo en el que está corriendo, y ajuste sus parámetros o ejecute subrutinas previamente optimizadas para la arquitectura de este [5]. Para el programador esto significa un enorme aumento de complejidad y tiempo de desarrollo/pruebas proporcionales a la variedad de dispositivos con los que se quiera trabajar.

Otro aspecto importante a tomar en cuenta es la mayor complejidad del API de OpenCL con respecto al CUDA Runtime API. Esto se debe a que el primero deja al programador la inicialización de los dispositivos que ejecutarán el código paralelo, mientras que el API de CUDA se encarga de esto de manera automática, al menos para implementaciones con una sola tarjeta con arquitectura CUDA. En caso de utilizar más de una tarjeta o necesitar acceso a todas las características de las mismas deberá usarse el CUDA Driver API, cuya complejidad es comparable a la de OpenCL. El estudio de este tipo de desarrollos escapa al alcance del presente documento que se centrará enteramente en el CUDA Runtime API, de menor complejidad.

Es apropiado mencionar que la *performance* computacional de OpenCL aún está por debajo del de CUDA [4], entre otras razones, por la relativa novedad de la primera con respecto a esta última tecnología y, como ya se mencionó, por el hecho que OpenCL toma en cuenta una gama diversa de dispositivos.

Esta diferencia puede apreciarse en la Figura 1, en la que se comparan implementaciones optimizadas del algoritmo Adiabatic QUantum Algorithms (AQUA) en CUDA y OpenCL en una tarjeta gráfica NVIDIA GeForce GTX-260. En esta se muestran los resultados de ambos para diferentes volúmenes de datos o cantidad variables procesadas.

Resulta pues una interesante alternativa a tener en cuenta en el futuro, ya que incluso NVIDIA ha mostrado su apoyo al estándar OpenCL [15], y se considera que un programador de CUDA no debe invertir esfuerzo considerable en portar su código de CUDA a este framework [5]. Por las razones expuestas se determina que CUDA es una opción sencilla y madura con la que pueden desarrollarse programas altamente paralelos. Así, el posterior análisis se centrará en este método.

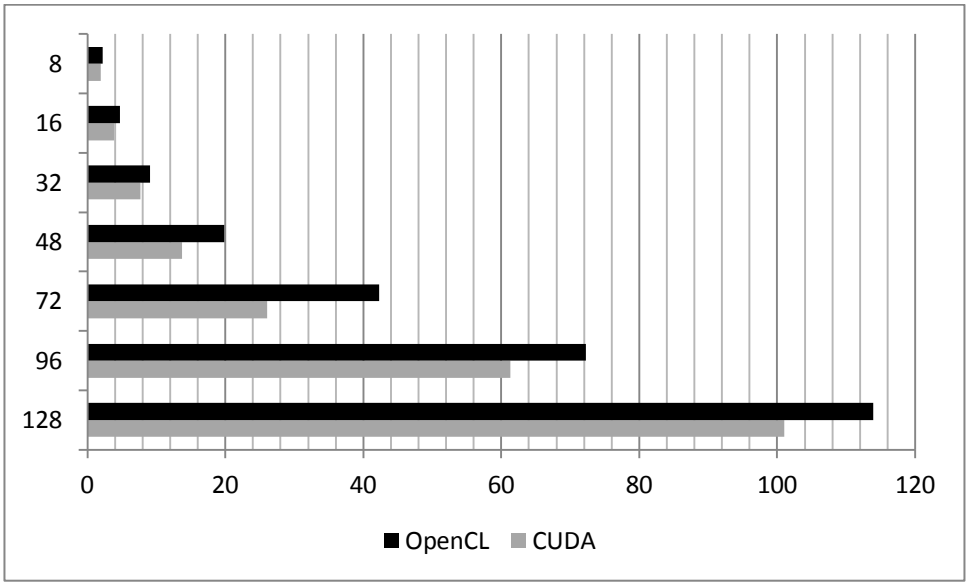

Figura 1. Tiempos de ejecución (en segundos) del algoritmo AQUA implementado en CUDA y OpenCL para distintos volúmenes de datos (8, 16, 32 ...,). Se aprecia como en todos los casos se obtienen tiempos de ejecución menores en CUDA. Tomado de [4].

Capítulo 2
Introducción

2.1. GPGPU (*General Purpose GPU*)

Hace relativamente poco (2007) NVIDIA publicó su SDK de CUDA. Este permite a los desarrolladores de software utilizar la capacidad multiprocesamiento de las tarjetas gráficas con arquitectura del mismo nombre para tareas de propósito general. Esto se conoce como GPGPU (*General Purpose Graphics Processing Unit*) y la idea consiste en utilizar la alta concurrencia de las tarjetas de video para aplicaciones de cómputo general como pueden ser simulaciones de tráfico [4], simulaciones físicas, algoritmos de búsqueda, cálculos con vectores o matrices de grandes tamaños, comparación de secuencias, entre otros.

Si bien la idea de utilizar procesadores gráficos para tareas generales ya existía antes de la aparición de CUDA, este entorno presenta ciertas ventajas frente a los enfoques antiguos como lo son los lenguajes como CG [20], APIs como OpenGL [18] e incluso frameworks como BrookGPU [19].

Entre las ventajas encontramos principalmente la capacidad de acceder a la memoria compartida de las tarjetas gráficas, la cual alcanza velocidades mucho mayores que la memoria del dispositivo y más importante, la posibilidad de trabajar en contextos no gráficos.

Sin embargo la más notoria desventaja de CUDA es que sólo se puede utilizar en GPUs de NVIDIA, por lo que la portabilidad de las aplicaciones desarrollas para esta arquitectura está claramente limitada a las tarjetas gráficas de este vendedor.

2.1.1. Arquitectura CUDA

La arquitectura de las tarjetas gráficas que cuentan con CUDA difiere de la de las tarjetas gráficas tradicionales principalmente en que está optimizada no sólo para aplicaciones gráficas. Las tarjetas de CUDA de NVIDIA se han desarrollado tomando en cuenta la creación de aplicaciones de propósito general y se pretende hacer del modelo de programación lo más transparente posible para el desarrollador. Además, una de las características principales es la posibilidad de trabajar sobre un modelo escalable que permita a los programas aprovechar las capacidades de las nuevas tarjetas sin grandes modificaciones.

Para comprender mejor la arquitectura CUDA es imprescindible dejar en claro algunos conceptos descritos a continuación.

Un *thread* o hilo es la unidad de ejecución en la que se procesa parte de la data y que puede sincronizarse y compartir información con otros hilos. Un *thread* en CUDA es similar a los *threads* en cualquier sistema operativo multitarea con la gran diferencia que la creación y sincronización de estos no supone un alto costo computacional como si lo hace en el caso de estos sistemas operativos. La principal razón radica en el modelo de ejecución de la arquitectura CUDA, SIMT (*Single Instruction Multiple Thread*) según el cual una misma instrucción es ejecutada por varios *threads* en simultáneo, por lo que el costo de ejecución se torna casi despreciable [5], es decir la arquitectura de la tarjeta de video está optimizada para realizar tareas de este tipo.

De esta forma, en CUDA resulta natural el uso de estas estructuras de ejecución mientras que en otras plataformas es necesario evaluar si vale la pena segmentar la ejecución de un programa o no tomando en cuenta el costo y la complejidad de crearlas y sincronizarlas

Para mayor detalle cabe aclarar que los *threads* se agrupan en bloques, que son una abstracción para denotar un arreglo (de hasta 3 dimensiones) de estos. Dichos bloques se agrupan, a su vez, en un *grid* que es un arreglo (también de hasta 3 dimensiones) de bloques. Esto se puede apreciar mejor en la Figura 2, en la cual además se observa el modelo de memorias de la arquitectura CUDA.

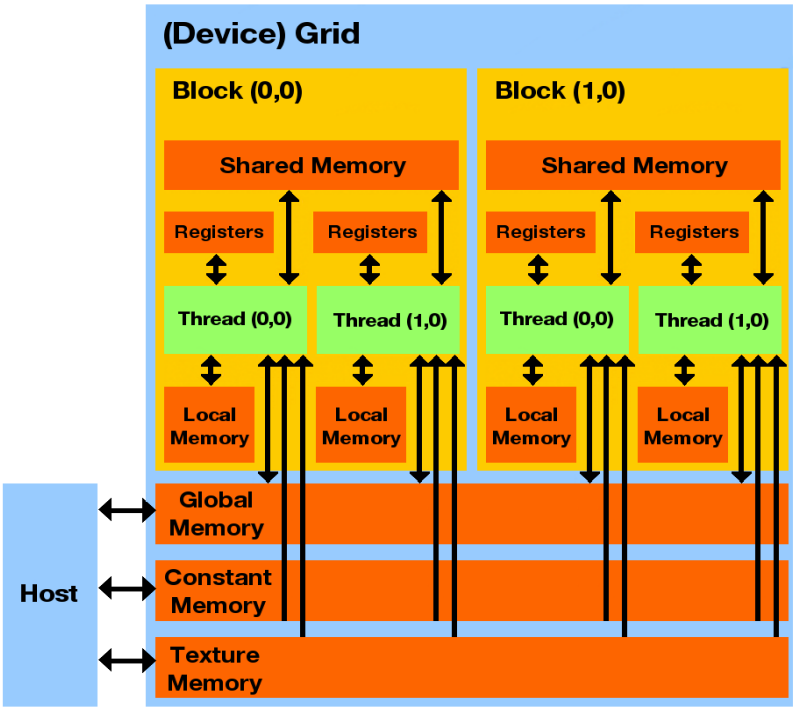

Figura 2: Varias memorias disponibles en la arquitectura CUDA (Imagen original tomada de http://www.ks.uiuc.edu)

Dicho modelo es relativamente complejo pues abarca diversos tipos de memorias y su comprensión es vital para el desarrollo de implementaciones óptimas. Más adelante en esta sección se verá cómo utilizar una u otra memoria puede afectar considerablemente la velocidad de un programa. Para un resumen de las diferentes memorias y sus políticas de acceso, ver Tabla 1.

De esta forma, se pueden listar la memoria constante y la memoria global que, siendo las de mayor tamaño, son también las más lentas. Ambas son compartidas por todo el dispositivo, por lo que serán referidas en lo que resta

del documento como memorias de dispositivo. Dentro de este grupo también se encuentra la memoria local. Esta, no es más que una porción de memoria global cuyo acceso se encuentra restringido a un solo *thread.* Consecuentemente, su velocidad es comparable a la del resto de memorias de dispositivo, esto es, relativamente lenta.

Por otro lado se encuentra la memoria compartida (shared memory en la Figura 2) a la cual se puede acceder por bloque, es decir, si se inicializa una variable en este tipo de memoria, cada bloque tendrá su propia versión. Teóricamente, el acceso a datos en este tipo de memoria es bastante más rápido que los accesos a memoria de dispositivo. Incluso se hace un símil de esta con la memoria caché de nivel 1 (*L1 Cache*) de los procesadores tradicionales [8]. Finalmente, como el tipo de memoria de mayor velocidad y menor tamaño, se tienen los registros que se considera trabajan a velocidad de reloj [6].

Tipo de Memoria	Acceso	Permisos	Velocidad
Registros	Por Thread	Leer/Escribir	Rápida
Shared Memory	Por Bloque	Leer/Escribir	Media
Global Memory	Por Grid	Leer/Escribir	Lenta
Constant Memory	Por Grid	Leer	Lenta
Texture Memory	Por Grid	Leer	Lenta (Tiene caché)

Tabla 1: Tipos de memoria y sus políticas de acceso.

Si se tiene presente que las tarjetas con arquitectura CUDA son finalmente dispositivos de procesamiento de video, no sorprenderá el hecho que la memoria de Textura o Texture Memory de la Figura 2 está optimizada para procesar imágenes y superficies tridimensionales. Es posible, por lo tanto, sacar provecho de la misma en ciertas aplicaciones en las que la localidad de los datos se asemeja a la de los pixeles de una imagen [8]. La ventaja de usar este tipo de memoria es que, a diferencia de las memorias global y constante, cuenta con un caché de datos.

Las características de la caché de texturas de CUDA difieren de las de las memorias caché tradicionales debido a que en el primer caso se toma en cuenta si se trabaja con arreglos de hasta 3 dimensiones, como se ilustra en la Figura 3. En esta se observa que cuando se trabaja con una memoria caché tradicional y se debe leer de memoria de dispositivo, se copia un bloque de memoria contigua para favorecer la localidad de datos en los siguientes accesos.

En el caso de la caché de textura de CUDA el proceso es casi el mismo, con la salvedad que en caso de trabajar con texturas bidimensionales o tridimensionales, el bloque de datos copiados a memoria caché será dimensionalmente consistente con la textura.

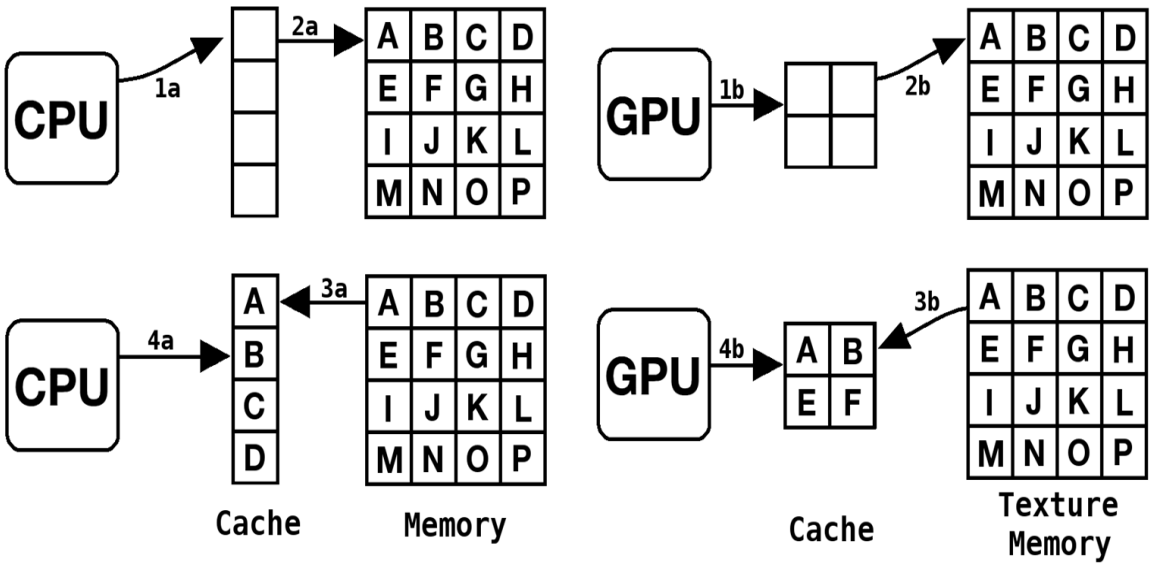

Figura 3: Accesos a memoria con caché tradicional (a) y caché de 2D en CUDA (b)

Se muestran los accesos a memoria (1), en los que se debe recurrir a memoria de dispositivo (2) por no encontrarse los datos en caché. Luego del primer acceso se copian los datos contiguos (3), encontrándose disponibles para el siguiente acceso (4). Nótese que en el caso 3a se copia un bloque de memoria linealmente contigua mientras que en el caso 3b se toma en cuenta la estructura bidimensional de la textura, copiándose valores espacialmente contiguos.

2.1.2. Ejecución de un Kernel

Se observa (Figura 4) que la declaración de un kernel es prácticamente la misma que la de una función en C con la diferencia que se utiliza la declaración __global__ para especificar que se trata de una porción de código que habrá de ejecutarse en el GPU. Por otra parte, además de pasar parámetros al kernel como si de una función se tratara, se especifican los parámetros de ejecución del mismo entre las llaves <<<...>>>.

```
// Declaración de un kernel para multiplicar los
// elementos de dos vectores

__global__ void kernel0(
     float *a,
     float *b,
     float *c,
     int N)
{
     int idx = threadIdx.x;
     c[idx] = a[idx]*b[idx];
}

// Llamada el Kernel

int main(){
     kernel0<<<Blocks,Threads,SMem>>>(
               vector_a,
               vector_b,
               vector_c,
               N);
 }
```

Figura 4: Declaración y llamada de un kernel CUDA para multiplicar dos arreglos punto a punto.

Con esto en consideración se puede comprender mejor la ejecución de un programa en la arquitectura CUDA. Como se observa en la Figura 5, la aplicación se ejecuta como un programa regular en el CPU (*Host*) hasta que se llega a la llamada al Kernel0, momento en el que se pasan los parámetros al kernel y este comienza a ejecutarse en el GPU (Device). Mientras sucede esto, el CPU está habilitado para continuar ejecutando las siguientes instrucciones del programa.

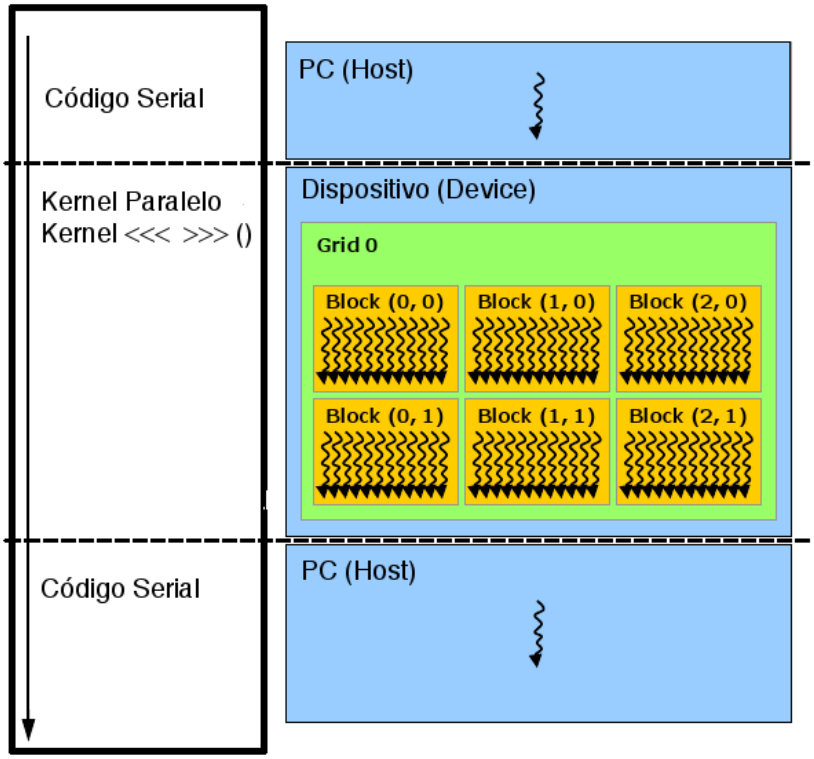

Figura 5: Flujo de la ejecución de un programa en la arquitectura CUDA (Imagen tomada de [8])

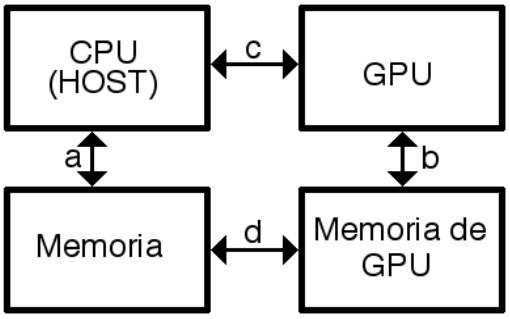

Figura 6: Relación entre los componentes de un sistema típico con un GPU CUDA.

Para finalmente comprender este proceso es apropiado relacionarlo con el hardware. Para esto se puede observar en la Figura 6 la relación entre los componentes del sistema en el que se encuentre instalada la o las tarjetas de video CUDA. De la misma forma que para la ejecución de un programa tradicional el CPU del computador lee sus instrucciones y datos de memoria

(Figura 6a), una tarjeta de video cuenta con un GPU (por cuestiones didácticas se simplifica el hecho que las tarjetas tienen varias unidades de procesamiento) que lee datos e instrucciones de su memoria de video (b). Para poder hacer esto el CPU debe inicializar los datos y parámetros del kernel que se ejecutará en el GPU (c) y estos deben ser copiados de la memoria de la PC a la tarjeta de video (d).

2.2. Conceptos Generales

En esta sección se presentan algunos conceptos necesarios a tener en cuenta para la comprensión del resto de este documento.

2.2.1 Representación de imágenes

Para representar una imagen de manera digital, es necesario hacerlo de manera discreta. Si bien existen varios tipos de representaciones, esta tesis se centrará en la espacial, cuyo uso es bastante común [3].

En esta, una imagen continua se aproxima como un arreglo numérico bidimensional, en el que los valores de cada punto se denominan pixeles. Es decir, se puede representar una imagen como una función de la forma $f(x, y)$ donde 'x' e 'y' son las coordenadas de sus píxeles como se muestra en la Figura 7 y $f(x, y)$ es el valor de dicho pixel. Este valor denota la irradiancia de la imagen en determinada posición. A mayor cantidad de pixeles se tendrá una representación más fiel de la imagen, esta cantidad se conoce como resolución. Así, se dice que una imagen de 640 píxeles de ancho y 480 tiene una resolución de 480x640. Ver figura 8.

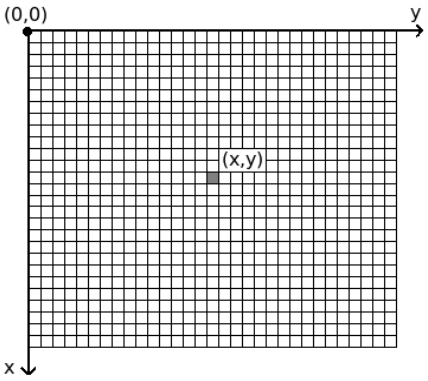

Figura 7: Representación de los píxeles de una imagen

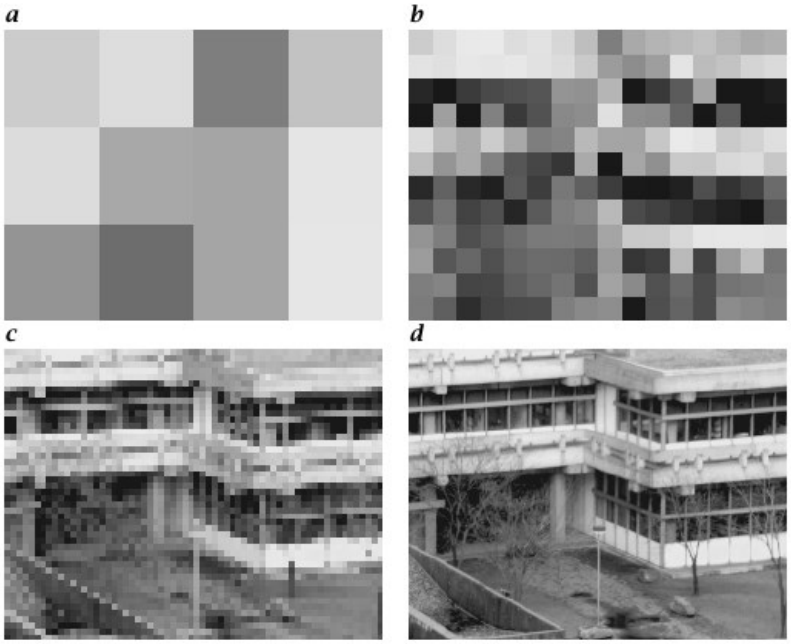

Figura 8: Imagen a distintas resoluciones. a) 3×4. b) 12×16. c) 48×64 d) 192×256 pixels. (Imagen tomada de [3])

Para representar una imagen es útil, además, tener una convención sobre la representación de los colores de la misma. Existen diversos modelos o sistemas de representación de colores, entre los que cabe destacar los más conocidos RGB, CMY, CMYK, HSI. Por ejemplo, en el modelo RGB de 24 bits cada valor de $f(x, y)$ contiene 3 enteros de 1 byte cada uno, con cada uno de los cuales se puede representar un número del 0 al 255.

Cada uno de estos 3 bytes simboliza la presencia de los colores rojo, verde y azul de manera respectiva, siendo 255 la máxima presencia del mismo y 0 su ausencia. De esta forma, pueden diferenciarse 2^{24} colores distintos.

En la presente tesis se trabajará principalmente con imágenes en escala de grises, por lo que es pertinente mencionar que para representar imágenes de este tipo se usa 1 byte por píxel (enteros sin signo), cuyos posibles valores van desde 0 a 255 pasando por un rango de grises. El 0 representa la ausencia de irradiancia y por tanto se interpreta como el color negro mientras que el nivel máximo (255 en este caso) representará al color blanco.

2.2.2. Convolución

Antes de definir el proceso de convolución resulta necesario dejar claros algunos conceptos. De la misma forma que se representa una imagen como una matriz bidimensional, existe una estructura llamada filtro espacial. Dicha estructura no es más que una matriz de NxM, cuyos valores se denominan coeficientes del filtro. Para efectos de facilitar los cálculos se considerará que N y M son siempre impares aunque en realidad pueden tener dimensiones arbitrarias. A lo largo del presente documento se utilizará el nombre filtro o máscara para hacer referencia a filtros espaciales.

Es posible diferenciar entre filtros lineales y no lineales [14]. Los primeros son como el mostrado en la Figura 9b2. Este al ser aplicado a una porción de imagen se obtiene la suma de productos de cada valor de dicha porción con cada valor del filtro. Es decir se trata de una operación lineal. Los filtros no lineales no suelen representarse como matrices y, como su nombre lo dice, la cantidad resultante se obtiene mediante una operación no lineal con una región de la imagen.

Así, por ejemplo, es posible tener un filtro no lineal en el que el resultado sea el mínimo o máximo valor de una porción de la imagen. Para el presente análisis y desarrollo se considerarán únicamente los filtros lineales. En a) se puede observar la imagen a filtrar y el filtro b2) respectivo. En el ejemplo, se toma la porción de la imagen b1) alrededor del pixel a filtrar (Valor 7) y se multiplica punto a punto para hallar la matriz c) resultante. En seguida se calcula la suma de los valores de dicha matriz y se obtiene el valor del pixel filtrado (Valor 4) cuyo valor se encuentra en la misma posición de la imagen filtrada que en la que se encontraba en la imagen original.

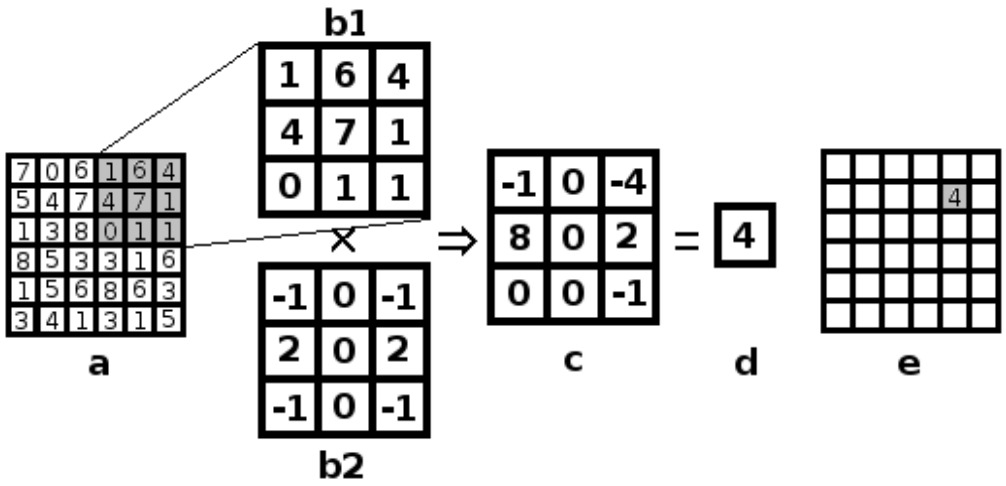

Figura 9: Aplicación de un filtro de 3x3 a un píxel de una imagen en escala de grises.

a) Imagen original. b1) Porción de la imagen a filtrar. b2) Filtro. c) Multiplicación punto a punto. d) Suma. e) Imagen filtrada.

Se observa, por lo tanto, que para calcular un pixel determinado debe tomarse una porción de la imagen original del mismo tamaño que el filtro a aplicar y realizar una cantidad de multiplicaciones igual a las dimensiones del filtro (9 en el ejemplo). Si se repiten estos pasos para cada pixel de la imagen, se dice que se ha aplicado el filtro espacial a la imagen, y el resultado será la imagen filtrada. Este proceso se conoce como convolución por lo que puede decirse que la convolución es el proceso por el cual se mueve un filtro sobre una imagen y se calcula la suma de los productos en cada posición [14].

Para comprender mejor este procedimiento es apropiado exponerlo en su notación matemática. Sea I una imagen y F un filtro de NxM se expresa la convolución de ambos como:

$$R(x,y) = \sum_u \sum_v I(x-u, y-v) \times F(u,v) \qquad (2.1)$$

donde

$$u \in [-\frac{(N-1)}{2}; \frac{(N-1)}{2}] v \in [-\frac{(M-1)}{2}; \frac{(M-1)}{2}] \qquad (2.2)$$

Es importante mencionar la cantidad de operaciones realizadas por este procedimiento, pues esta determinará en gran medida el desempeño de las implementaciones. Por simple inspección se nota que la cantidad de multiplicaciones para una matriz de NxM y un filtro de AxB será AxBxNxM. De manera similar, en el caso separable se tendrán (A+B)xNxM multiplicaciones. El cálculo de estos valores se verá en detalle en las secciones 3.3.1 y 3.3.3 del siguiente capítulo respectivamente.

No queda claro qué sucede en los bordes de las imágenes, pues cuando se centre el filtro en los mismos, parte de este caerá fuera de aquella. Existen diferentes métodos para solucionar este problema y todos ellos introducen algún tipo de error [3]. El primero que se mencionará consiste en tomar la imagen procesada como un “mosaico”, lo que se conoce como convolución cíclica, de esta manera, cuando se llegue por ejemplo al extremo derecho de la imagen, el siguiente valor será el primer valor a la izquierda en la fila correspondiente. Otro método consiste en tomar como 0 todos los valores que se encuentren fuera de la imagen, o extrapolar los valores de los bordes, repitiéndolos fuera de la imagen. Debido a que en todos los casos se introduce algún tipo de error, se escogerá tomar como 0 todos los valores fuera de la imagen por cuestión de simplicidad [3].

La importancia de la convolución radica en su utilización extensiva en varias áreas del procesamiento digital de imágenes, siendo la aplicación de filtros espaciales la más notable. Dependiendo de la aplicación, pueden utilizarse

filtros especialmente diseñados para resaltar alguna de las características de una imagen (detalles, bordes horizontales o verticales, cambios bruscos) o para suavizar las mismas. En general, estas técnicas pueden utilizarse para la reconstrucción de imágenes.

Capítulo 3:
Desarrollo e Implementación

3.1. Objetivos

La presente tesis consiste en la implementación de la operación de convolución en dos dimensiones en la arquitectura multiprocesador CUDA. Los objetivos de esta investigación comprenden la implementación de funciones generales para la ejecución de la convolución en tarjetas gráficas basadas en la arquitectura CUDA. Para esto, cabe mencionar que se tomarán en cuenta dos casos particulares. El primero es el proceso de convolución descrito en la sección 2.2 del capítulo 2 y el segundo es el caso en el que por las características del filtro a aplicar se puede reducir la cantidad de operaciones realizadas. En este último caso se dice que el filtro es separable y su análisis se hará en la sección 3.3.3 del presente capítulo.

Por otro lado, se compararán estas implementaciones con otras, entre ellas las lineales y las paralelas, incluso las del propio SDK de CUDA. El objeto de esto será medir cuantitativamente el aumento en desempeño de las soluciones planteadas y determinar bajo qué condiciones se aprovecha mejor la implementación paralela.

3.2. Metodología

Para cumplir los objetivos establecidos previamente, se hará uso del framework oficial de NVIDIA. Así, se desarrollará en el lenguaje C extendido por las funciones de la librería “cuda.h”. Inicialmente se implementará el algoritmo de forma lineal y luego se reescribirá la “parte crítica” (bucles principales) en CUDA.

Además se programará teniendo en cuenta la capacidad de escoger con cuantos *threads*, bloques y con qué cantidad de memoria compartida se invocarán los kernels así como el tamaño de la matriz con que se trabajará. Esto permitirá realizar pruebas para distintos tipos de parámetros de los kernels y de esta forma se tendrá una aplicación flexible para realizar distintos tipos de mediciones.

3.2.1 Entorno de Desarrollo

Se trabajará enteramente en un entorno de desarrollo Linux utilizando herramientas del sistema para la programación y compilación de los programas. Para este último propósito se utilizará nvcc (*NVIDIA CUDA Compiler Driver*), el compilador que provee el NVIDIA CUDA Toolkit. No obstante, el código producido puede ser compilado bajo otras arquitecturas y sistemas operativos siempre que se cuente con hardware con arquitectura CUDA o un emulador de la misma. Las características de la PC en la que se trabajará, cuya importancia radica en el hecho que se tomará en cuenta al momento de medir el tiempo de ejecución de implementaciones ajenas a CUDA, se muestran a continuación:

Procesador: Intel Core i5 M520 2.4GHz.
Memoria: 6GiB de memoria RAM.
Tarjeta de Video: NVIDIA GeForce 330M con memoria de video dedicada de 1GiB. (Ver más detalles en Anexo B).
Driver NVIDIA: El driver de la tarjeta de video es la versión 270.41.06 para el sistema operativo linux de 64 bits.

Es posible estimar de manera aproximada el rendimiento teórico de la tarjeta en que se trabajará. Esta cuenta con 6 multiprocesadores, con 8 procesadores o cores cada uno, para un total de 48 unidades de procesamiento. Cada una de estas cuenta con un reloj interno de 1.265Ghz con lo que se calcula un

rendimiento teórico de 60.72Gflops. Esto es 60.72 miles de millones de operaciones en punto flotante por segundo.

En la práctica se espera un rendimiento mucho menor pues se deben tener en cuenta los accesos a memoria y las operaciones secundarias para cálculo de índices, valores temporales de registros y lectura de instrucciones entre otras. Es decir, la capacidad de cómputo teórica nunca será completamente aprovechada para los cálculos específicos del algoritmo desarrollado. Cuando se analicen los resultados de las implementaciones en el capítulo 4, se hablará de Gflops efectivos cuyo valor se aproximará a la cantidad de multiplicaciones realizadas.

3.2.2 Librerías externas

La única librería externa a la librería C estándar que se utilizará es *cuda.h* de NVIDIA cuya necesidad es indiscutible, pues permitirá utilizar las directivas y funciones para sacar provecho a la tarjeta de video. Estas van desde copias de memoria entre *host* y dispositivo hasta sincronización de threads.

3.2.3 Mediciones de tiempo y benchmarking

Para la medición precisa de los tiempos de ejecución de los kernels y de las copias de datos de memoria de CPU a memoria de dispositivo y viceversa se utilizará NVIDIA Compute Visual Profiler versión 3.2 [7]. Esta herramienta permite determinar los tiempos de ejecución de los kernels y las copias de memoria, así como determinar el porcentaje de uso de la capacidad de cómputo de la tarjeta de video, la cantidad de memoria utilizada para cada tipo de memoria de la tarjeta gráfica, entre otras mediciones útiles para la optimización de los programas implementados [7].

Cabe mencionar que también se realizaron pruebas de tiempo de ejecución de programas enteros, esto es, los kernels más las subrutinas que generan matrices aleatorias. La finalidad de estas pruebas fue hacer una comparación rápida entre varias implementaciones considerando que las subrutinas ajenas al kernel tienen un tiempo de ejecución constante. Para esto se desarrolló una herramienta en python para medir el tiempo de ejecución de programas arbitrarios. Ver Anexo A.

3.3. Propuesta

La idea general del programa a desarrollar es tomar una matriz y un filtro de entrada y entregar dicha matriz convolucionada con este, como se muestra en la Figura 10. A fin de mantener la implementación lo más general posible, se realizará este proceso para arreglos bidimensionales de valores con signo en punto flotante, siendo las imágenes en escala de grises (enteros entre 0 y 255) un subgrupo de este tipo de valores.

Figura 10: Proceso general a desarrollar

Por otro lado, esto aprovecha el hecho de que en la arquitectura CUDA pueden conseguirse mayores prestaciones si se trabaja con valores en punto flotante [8], debido a que su hardware cuenta con unidades de punto flotante dedicadas de simple precisión (2 unidades para dispositivos con versión 1.X, 4 para dispositivos 2.0 y 8 para dispositivos a partir de la revisión 2.1) y una unidad de punto flotante doble precisión [8]. De esta forma, el tiempo de

ejecución de operaciones sobre este tipo de datos se verán beneficiada por las mencionadas particularidades de la arquitectura.

3.3.1. Algoritmo propuesto

La primera aproximación para resolver el problema de la convolución es iterar a lo largo de cada valor de la matriz de entrada y aplicar el filtro a dicho valor y sus alrededores, obteniendo el valor respectivo de la matriz de salida. Esto es, aplicar el procedimiento mostrado previamente en la Figura 9 a cada elemento del arreglo bidimensional. A continuación se muestra el pseudocódigo simplificado de esta implementación.

```
Para cada x,y en Entrada
    Para cada i,j en Filtro
        Salida[y][x] += Entrada[y+j][x+j] * Filtro[j][i]
```

Figura 11: Pseudocódigo de la implementación simple.

Se puede notar que este procedimiento es una aplicación directa de la fórmula 2.1. mostrada anteriormente. Si la matriz de entrada tiene dimensiones NxM y el filtro AxB, se deberán realizar NxMxAxB multiplicaciones secuenciales.

Por simple inspección se nota que este algoritmo es altamente paralelizable, pues los cálculos de determinado elemento de la matriz de entrada son completamente independientes de los cálculos de cualquier otro elemento. Esto es, determinar el resultado de aplicar el filtro al valor con posición (x,y) no depende en absoluto del resultado de aplicar el filtro al elemento con posición (x+a,y+b). Este paralelismo un paralelismo del tipo independiente.

Esto lleva a la implementación de un primer kernel en CUDA. La idea es utilizar una cantidad de *threads* igual a la cantidad de elementos de la matriz a procesar. Entonces, cada *thread* puede encargarse de un valor de la matriz e

iterar sobre todos los valores del filtro, consiguiendo reducir la ejecución a sólo AxB multiplicaciones secuenciales, lo cual idealmente llevaría a reducir el tiempo de ejecución del kernel a lo que demore realizar estas AxB multiplicaciones.

Sin embargo, en la realidad esto no sucede así sobre todo porque si bien las tarjetas con arquitectura CUDA pueden ejecutar una gran cantidad de *threads* en simultáneo (8 *threads* por multiprocesador) es completamente factible trabajar con matrices de al menos 1024x1024 elementos, lo que implicaría trabajar con poco más de un millón de *threads*. Por lo tanto, se necesitarían (1024x1024)/8=131072 multiprocesadores, lo que sobrepasa enormemente la capacidades de las mejores tarjetas CUDA disponibles en el mercado.

Lo que sucede en realidad es que los bloques de *threads* se distribuyen en los multiprocesadores, dentro de cada uno de los cuales se dividen en grupos de 32 *threads* (1 *warp*) que son ejecutados de forma casi simultánea, pues cada multiprocesador cuenta con 8 procesadores físicos independientes. Por esta razón, un warp se ejecutará en grupos de 8 *threads*, es decir, cada procesador ejecutará 4 *threads* de un mismo warp de manera secuencial como se explica en [13]. Los *threads* restantes se ejecutarán en warps de manera secuencial.

Finalmente se obtendrá un tiempo de ejecución considerablemente menor al que idealmente se esperaba, pues no todos los *threads* se ejecutan en el mismo instante.

3.3.2. Uso de Memoria Compartida

Un aspecto que limita el desempeño de los kernels de CUDA son los accesos a memoria. Es conocido el hecho que para los accesos a memoria local sin caché o memoria global se pueden esperar latencias de entre 400 a 600 ciclos de

reloj. Por esta razón la optimización, léase minimización, de accesos a memoria se considera de alta prioridad al momento de optimizar un kernel CUDA [6].
Debe tratar de utilizarse memoria compartida (shared memory) siempre que se pueda y en casos en que las capacidades de la misma (16KB para dispositivos 1.X y 48KB para dispositivos a partir de 2.0) [8] se vean sobrepasadas por la cantidad de datos resulta conveniente reordenar y agrupar los accesos a memoria o considerar el uso de memoria de textura si la aplicación lo permite. El uso de esta última debe preferirse siempre que sea posible sobre el uso de memoria global o constante, principalmente por el hecho que a diferencia de las últimas, esta cuenta con una memoria caché.

En el caso del algoritmo propuesto, se utilizará memoria compartida o shared memory para guardar los datos calculados. Estos valores son un claro candidato para el uso de este tipo de memoria pues varían constantemente y son accesados varias veces por cada pixel calculado (recordar que para calcular cada pixel de salida deben multiplicarse y sumarse una cantidad de valores iguales al tamaño del filtro utilizado).

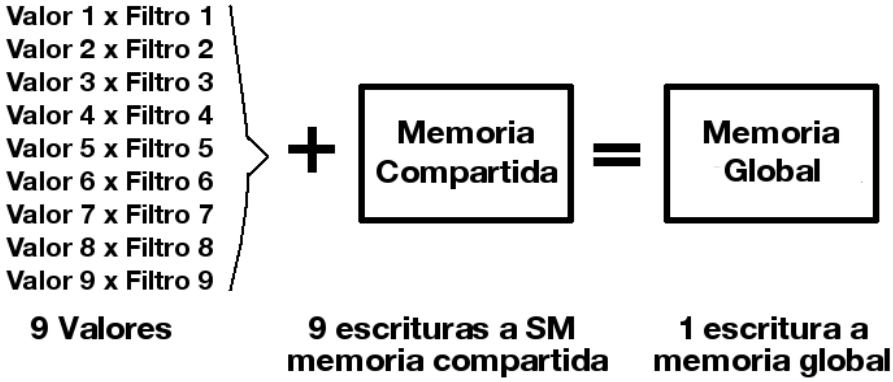

Figura 12: Para calcular un pixel de salida con un filtro de 3x3 se deberá sumar sucesivamente 9 valores en memoria compartida. Sólo cuando ya se tiene el valor final se escribe el mismo en memoria global.

Con esta implementación se evitará tener que escribir el resultado de cada multiplicación en memoria global (matriz resultado), pues sólo se escribirá la suma de estas. Así, por ejemplo, para un filtro de 3x3 se evitará escribir 9

valores en memoria global y por el contrario se escribirá sólo 1. La figura 12 describe este proceso de manera gráfica.

3.3.3. Algoritmo Separable

Otro método para disminuir el tiempo de procesamiento consiste en explotar el hecho que determinados filtros son separables. Esto es, pueden ser expresados como el producto externo de un vector fila y un vector columna (Figura 13), sin embargo, en este caso particular, la convolución bidimensional de los mismos dará el mismo resultado.

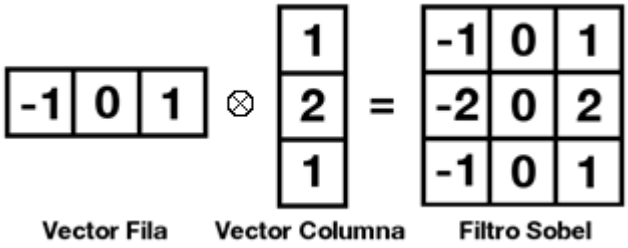

Figura 13: El filtro Sobel puede separarse en un vector fila y un vector columna.

Entonces se puede expresar la convolución de una imagen I por un filtro separable f como la convolución de dicha imagen con un vector fila y un vector columna como se muestra a continuación.

Se tiene $I * f = R$

Si $f = f_{col} * f_{fila}$

entonces $I * f_{col} * f_{fila} = R$

La ventaja de expresar la operación de esta manera es que permite reemplazar la convolución bidimensional por dos convoluciónes unidimensionales consecutivas, siendo estas últimas menos complejas que la primera.

Computacionalmente se realizan menos operaciones que en el caso no separable, pues si las dimensiones del filtro son AxB y las de la matriz NxM, deberán realizarse NxMxAxB multiplicaciones para obtener la convolución bidimensional con el método directo, mientras que el total de multiplicaciones realizadas usando el método separable será (A+B)xNxM y como se conoce que A+B<AxB para valores de A y B mayores a 1 se tendrá finalmente una cantidad menor de multiplicaciones, por lo que en teoría el tiempo de procesamiento debería ser menor.

En este caso el algoritmo para la convolución con el vector fila será el siguiente:

```
Para cada x,y en Entrada
  Para cada i en Filtro_Parcial
    Salida[y][x] += Entrada[y][x+i] * Filtro_Parcial[i]
```

Figura 14: Primera parte del algoritmo para convolución separable

y el correspondiente para el vector columna:

```
Para cada x,y en Entrada
  Para cada j en Filtro_Parcial
    Salida[y][x] += Entrada[y+j][x] * Filtro_Parcial[i]
```

Figura15: Segunda parte del algoritmo para convolución separable.

La implementación en CUDA se hará dividiendo la matriz original de manera similar a como se hace en el algoritmo no separable. Se separará la matriz en determinada cantidad de submatrices que serán procesadas por un número igual de CUDA *blocks*. Dentro de cada una de estas submatrices se tendrá una correspondencia de 1 a 1 entre los *threads* y la cantidad de elementos. De esta manera cada *thread* procesará un elemento determinado y sus valores contiguos según el tamaño del filtro vector.

Las operaciones de convolución con el vector columna y con el vector fila serán básicamente las mismas con una variación en los índices de las matrices. Por esta razón, se programará un kernel para el primer caso y se modificará apropiadamente para el segundo, con lo que se ejecutarán dos kernels de forma secuencial para calcular el resultado final.

En este caso también se utilizará memoria de textura tanto para la matriz como el filtro y memoria compartida para almacenar los resultados parciales de cada multiplicación y así reducir el número de escrituras a memoria global.

Sin embargo, por el hecho de hacer llamadas a dos kernels distintos se presentan algunas particularidades que resulta apropiado mencionar. En primer lugar, hay que evitar copiar los resultados al *host* una vez que el primer kernel ha terminado, pues estas copias de memoria implican un tiempo de ejecución considerable [6]

.

Por otro lado, el hecho de usar memoria de textura obliga a realizar una copia de memoria dentro del dispositivo. Esto se debe a que cuando se utiliza memoria de textura, lo que se hace es vincular una porción de memoria de dispositivo a un identificador de textura, esto hace que dicha porción sea de sólo lectura. Como el primer kernel debe escribir en un fragmento de memoria R, este no puede estar vinculado inicialmente a una textura. Sin embargo, el segundo kernel referenciará los valores de R mediante memoria de textura.

Por lo tanto, lo que se debe hacer es copiar los valores del resultado R a una porción de memoria temporal T, que si puede vincularse luego a una textura. Esta copia de memoria de dispositivo a memoria de dispositivo supone cierto tiempo de procesamiento que será medido y considerado en los resultados finales.

3.3.4. Optimizaciones Misceláneas

Adicionalmente, pueden mencionarse algunas optimizaciones al código de las implementaciones realizadas para obtener tiempos de ejecución ligeramente menores. A continuación se explicarán brevemente.

Dentro de los kernels utilizados tanto en la implementación del algoritmo propuesto y del algoritmo separable, cada *thread* ejecuta secuencialmente una serie de pasos, específicamente debe iterar sobre todos o algunos valores del filtro según sea el caso. Esto implica necesariamente el uso de iteraciones con un límite variable según las dimensiones del filtro utilizado.

Con este fin, deben restringirse las iteraciones en los casos en que parte del filtro se ubica fuera de la matriz, pues como se mencionó anteriormente se asumirá estos elementos como de valor 0. Esto es, no realizar ningún cálculo en estos casos.

Inicialmente para lograr esto se verificaba que el valor a calcular estuviera dentro de la matriz. Esto resultaba computacionalmente ineficiente pues la verificación debía repetirse en cada iteración. Ante esto, se modificó el código para realizar la verificación antes de empezar a iterar (fuera del bucle) y según sea apropiado, modificar el rango de iteración para no incluir los casos. Esto puede apreciarse mejor en la Figura 16.

Por otra parte, como se puede observar en el código mostrado en los anexos, la lectura a memoria de Textura se realizó mediante la función tex2D(), que recibe como parámetros el identificador de una textura a leer y las coordenadas (x,y) que se desea leer de la misma. Se observó que si los parámetros de las coordenadas a leer son del tipo punto flotante en vez de enteros el tiempo de ejecución de dicha función es ligeramente más bajo, como se mostrará en los resultados en el siguiente capítulo.

Caso Inicial

```
Para cada i entre (i_inicio;i_fin):
    Si (x+i) dentro de la matriz:
        //Realizar calculos...
```

Caso Optimizado

```
Si (x+i_fin) fuera de la matriz:
    i_fin = i_fin - restante
Para cada i entre (i_inicio;i_fin):
    //Realizar calculos...
```

Figura 16: Optimización de verificación de condiciones antes de la iteración

Por esta razón, si se convierten dichos índices a valores de punto flotante antes de utilizarlos en la función se conseguirá una disminución del tiempo de ejecución. Para lograr esto se deben declarar los índices 'x' e 'y' utilizados en tex2D(x,y) como valores de punto flotante siempre que sea posible, de manera que se acelere la operación de lectura de texturas.

Una última optimización será el uso de la instrucción __mul24() de CUDA. Esta calcula el producto de 2 valores enteros tratándolos como valores de 24 bits independientemente de que estos estén expresados como valores de 32 bits. Es decir, multiplica los 24 bits menos significativos de 2 valores enteros [8].

Esto es conveniente en algunos casos, debido a que en arquitecturas anteriores a la 2.0 las operaciones de multiplicación de enteros son compiladas como varias instrucciones, por lo que el uso de esta directiva cuando sea posible disminuirá la cantidad total de instrucciones a ejecutar [8]. Debe notarse, sin embargo, el hecho de que para valores enteros grandes que deben ser expresados con más de 24 bits, esta instrucción dará resultados erróneos sin producir un error en tiempo de compilación.

Capítulo 4:
Presentación y Análisis de Resultados

4.1. Resultados

En este capítulo se presentarán los resultados obtenidos al ejecutar las distintas implementaciones programadas. Se considerarán múltiples tamaños de matrices para analizar si se obtiene un aumento de *performance* o no.

Se compararán las funciones desarrolladas con las implementaciones del SDK de CUDA [21], específicamente con la de convolución bidimensional basada en la transformada de Fourier (*convFFT2D*) y la separable basada en memoria de textura (*convTexture*). Adicionalmente se medirán los tiempos de ejecución de la función conv2 de Matlab que implementa la misma operación y finalmente los del IPP Convolution Toolbox [23] basado en OpenMP y optimizado para procesadores Intel mediante directivas IPP [22].

4.1.1. Rendimiento de las implementaciones

Como ya se mencionó se utilizará el CUDA Compute Profiler de NVIDIA para medir el tiempo de ejecución de las implementaciones así como de las varias copias entre memoria de *host* y memoria de dispositivo.

Empezando por la implementación más lenta (Tabla 2) se tiene la del algoritmo propuesto sin ningún tipo de optimización (Apéndice C.1). Nótese que en la tabla mostrada se tienen los tiempos de ejecución de los kernels CUDA sin tomar en cuenta las copias de memoria, pues lo que se pretende es comparar las mejoras de las implementaciones.

	Tiempo (us)		
	1024x1024	**2048x2048**	**4096x4906**
Algoritmo Simple	6537	26075.9	104280

Tabla 2: Tiempos de ejecución en microsegundos del kernel sin optimizaciones para un filtro de 3x3 y matrices de varios tamaños.

Por el momento no se comparará con otras implementaciones fuera de CUDA para centrar el análisis en las optimizaciones realizadas. Sin embargo, resulta necesario poner en contexto esto resultados.

Se calculará la performance computacional aproximándola a la cantidad de operaciones matemáticas de multiplicación que se deben realizar para obtener el resultado. Esta se expresará en Gflops (miles de millones de operaciones en punto flotante por segundo). Como se mencionó en el capítulo 3 el número de operaciones en la convolución bidimensional será AxBxNxM para una imagen de MxN y un filtro de AxB. En el caso separable, se tendrá (A+B)xNxM para un filtro e imagen de las mismas dimensiones.

En el primer caso, se calculó con una matriz de 1024x1024 y un filtro de 3x3, por lo que se realizaron más de 9.4 millones de multiplicaciones. Si para efectos prácticos no se consideran las operaciones de cálculo de índices, lecturas/escrituras a memoria y uso de registros, se calcula una performance computacional de aproximadamente 1.44Gflops efectivos. En los otros dos casos (2048x2048 y 4096x4096) se obtienen resultados casi idénticos.

A continuación, en las Figura 17, se muestran los resultados para las varias implementaciones probadas. La performance promedio es el promedio matemático de las performances de cada implementación ejecutadas para cada uno de los tres tamaños de matriz.

En la implementación 2 se utiliza memoria compartida (shared memory) para

guardar los datos de la forma descrita en la sección 3.3.2 del capítulo 3. Como se puede observar la mejora no es significativa aunque si apreciable. Es notable, sin embargo, el hecho que utilizar escrituras diferidas a través de memoria compartida en oposición a directas a memoria global brinde una mejora de rendimiento.

En el caso 3 ya se aprecia una mejora bastante significativa del rendimiento en un factor de 1.8 con respecto al caso 1 (sin optimizar). Esta mejora está basada en el hecho de utilizar memoria de textura tanto para la matriz como para el filtro y la consecuente explotación de la memoria caché de la misma.

	Implementación	Tiempo (us)			Performance promedio (Gflops)
		1024x1024	2048x2048	4096x4906	
1	**Algoritmo Simple**	6537	26075.9	104280	1.446
2	**Shared Memory (SM)**	6029	24592.5	98704.7	1.543
3	**SM+Texture Memory (TM)**	3509.3	13996.7	55942.8	2.695
4	**SM+TM+Reducir Loops (RL)**	2750	10969.2	43800.6	3.44
5	**SM+TM+RL+Misceláneas**	2262.4	9000.6	35954.5	4.188
6	**Algoritmo Separable**	1965.8	7818.2	31294.2	3.212

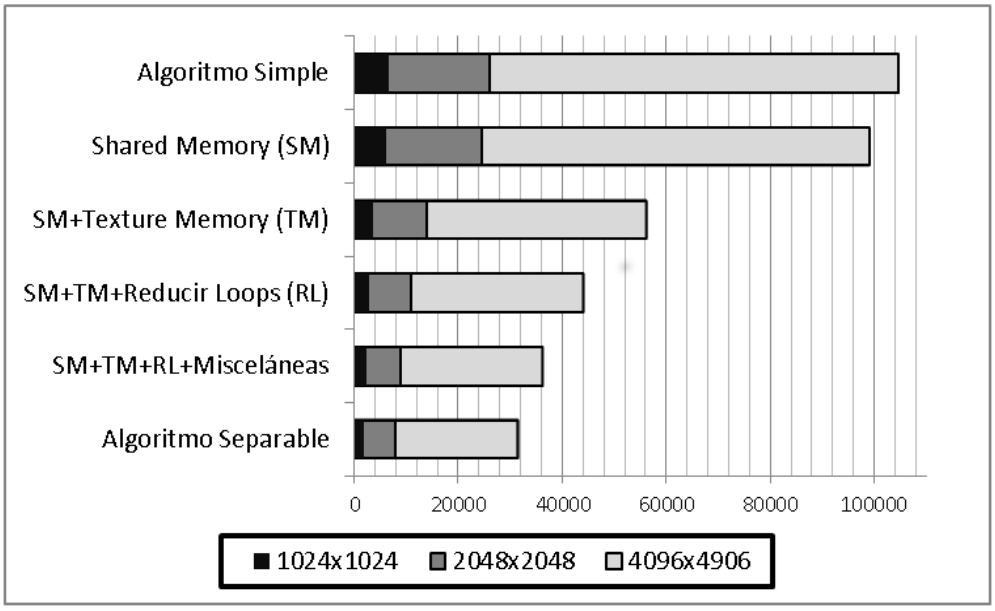

Figura 17: Tiempos de ejecución en microsegundos y rendimiento de las distintas implementaciones realizadas para varios tamaños de matrices.

Otra mejora importante se nota en el kernel 4, en el que se optimiza el bucle o loop que itera a través de los valores del filtro moviendo la verificación de encontrarse dentro de los bordes de la matriz fuera de dicho bucle. De esta forma, se evita repetir instrucciones innecesariamente, obteniéndose una reducción apreciable en el tiempo de ejecución. (Para más detalles ver apéndice C.2. y C.3.)

Finalmente, para el caso 5 se realizan lo que se ha denominado optimizaciones misceláneas. Se realizan dos cambios al código, el primero consiste en declarar los índices que se utilizarán para hacer referencia a las coordenadas de las texturas por valores en punto flotante en vez de enteros. Esto se estudió en la sección 3.3.4 del capítulo anterior.

El segundo cambio consistió en utilizar la instrucción intrínseca de CUDA __mul24 en el cálculo de los índices de la matriz resultado. De esta manera no se debe separar la multiplicación en varias instrucciones, aunque, como se mencionó anteriormente, esta optimización ya no es necesaria para dispositivos a partir de la versión 2.0, pues estos optimizan la multiplicación de enteros de manera automática. Con estos dos cambios simples se obtiene una mejora del rendimiento con respecto al caso 1 de aproximadamente 9 veces.

El caso del algoritmo separable tiene algunas particularidades que resulta necesario mencionar. En primer lugar, el cálculo del rendimiento difiere del de los demás casos principalmente porque en este se procesa la matriz dos veces. Sin embargo, cada vez que se hace esto se multiplica cada valor únicamente por una cantidad de valores igual al ancho o largo del filtro.

Así, por ejemplo, para el primer caso se multiplicará cada uno de los 1024x1024 elementos de la matriz por 3 elementos del vector columna y luego nuevamente por tres elementos del vector fila. Es decir, se tienen 1024x1024x3+1024x1024x3 multiplicaciones. Tomando en cuenta que el tiempo

mostrado en la tabla es la suma del tiempo de ejecución de ambos kernels se calcula, 3.212 Gflops efectivos. Cabe mencionarse, además, que en el resultado mostrado en la tabla no se ha considerado el tiempo de la copia de memoria entre la ejecución de los kernels mencionado en la sección 3.3.3 del capítulo 3.

Finalmente se presentan los resultados tomando en cuenta los tiempos de copia a memoria de *host* a dispositivo y viceversa y particularmente en el caso del algoritmo separable, el tiempo de copia de memoria dispositivo a dispositivo. Tomando en consideración dicho tiempo se obtienen los resultados mostrados en la Figura 18 (casos 1 y 2).

Implementación		Tiempo en microsegundos			
		1024x1024	**2048x2048**	**4096x4096**	**8192x8192**
1	**Separable Desarrollado**	7181	32984	157935	613762
2	**No separable Desarrollado**	7329	34763	151144	625786
3	**Matlab**	21800	99720	360390	1292300
4	**Intel IPP+OpenMP**	5600	19700	56426	154500
5	**convTexture SDK**	6989	29217	120011	476098
6	**convFFT2D SDK**	10970	55638	210585	sin datos

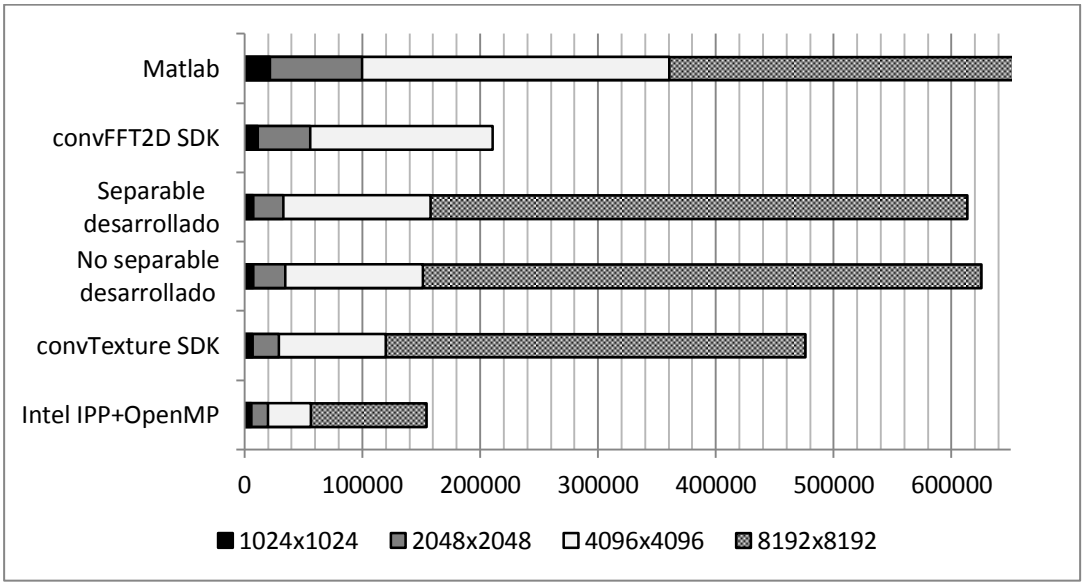

Figura 18: Tiempos de ejecución incluyendo copias de datos de varias implementaciones. (Casos 1 y 2 son las desarrolladas para la presente tesis)

Adicionalmente en esta figura, se muestran los resultados obtenidos con otras implementaciones ya existentes que resuelven el mismo problema, a manera de poner los resultados en contexto frente a otros programas disponibles. Para obtener una comparación justa se midió el tiempo entre la llamada a la función con los datos listos para ser procesados y la obtención del resultado.

Queda claro que la implementación más lenta de las analizadas es la del software matemático Matlab, específicamente su función conv2. Esto no resulta sorprendente pues este software debe soportar una extensa gama de hardware, por lo que su portabilidad no le permite realizar optimizaciones específicas para las arquitecturas utilizadas como se ha hecho en los demás casos.

Por otra parte, los casos 5 y 6 corresponden a funciones disponibles en el SDK de CUDA, siendo la 5 separable y las 6 no separable. Vale notar que la implementación desarrollada para el algoritmo separable es más lenta que la disponible en el SDK de CUDA (7181 vs 6989 microsegundos para la matriz de menor tamaño), no obstante, para el caso no separable se cuenta con una implementación con ligeramente mayores prestaciones que la de NVIDIA, para todos los casos estudiados. Además, puede observarse que para matrices de mayores tamaños (a partir de 4096x4096) el algoritmo no separable implementado presenta un tiempo de ejecución menor al separable debido al extenso tiempo de copia de memoria de dispositivo a dispositivo mencionada previamente.

Finalmente, cabe mencionar el caso 4, en el que se midieron los tiempos de ejecución de una solución paralela desarrollada por dos investigadores de NYU optimizada utilizando las librerías IPP (*Intel Performance Primitives*) y directivas OpenMP [23].

Esta implementación, si bien no toma en cuenta los valores de los bordes de la matriz (es equivalente a la opción 'valid' de conv2 en Matlab) y devuelve una

matriz resultado de menor tamaño que los demás programas, sirve para establecer la idea de que pueden conseguirse rendimientos similares a CUDA con métodos alternativos. Es bastante notable, sin embargo, que su tiempo de ejecución sea tan bajo, incluso comparado con las implementaciones del SDK de NVIDIA.

Conclusiones

Finalmente se concluye que CUDA brinda una interfaz sencilla para implementar algoritmos paralelos de manera eficiente. Esto se muestra por el hecho de haber conseguido implementar una solución aproximadamente 3 veces más eficiente que Matlab y de rendimiento similar a otras implementaciones paralelas altamente optimizadas. Si bien las implementaciones en OpenMP e IPP son más eficientes que las desarrolladas en CUDA, es apropiado atribuir los resultados al gran nivel de optimización de esta implementación y a las prestaciones del CPU utilizado.

Por otro lado, se nota que de los programas desarrollados, la versión separable es ligeramente más rápida que la no separable. Esto responde al hecho que para procesar una misma cantidad de valores, el primer algoritmo requiere menos operaciones matemáticas. Sin embargo, esta diferencia se hace menos evidente cuando se toman en cuenta los tiempos de copia de memoria entre la ejecución de los kernels para el primer caso. Finalmente ambas implementaciones tienen un tiempo de ejecución similar e incluso para matrices de 8192x8192 el segundo caso es más rápido. Por esta razón se considera que el caso separable no brinda una mejora en desempeño real con respecto al no separable, al menos no en las implementaciones realizadas.

A pesar de haberse implementado una operación altamente paralelizable, se puede afirmar en general que los resultados ponen en evidencia que la computación de propósito general en GPUs es una alternativa viable para la implementación de algoritmos paralelos eficientes de manera relativamente sencilla.

Recomendaciones

Existen varias recomendaciones para trabajos posteriores sobre el tema del presente documento.

Sobre los programas desarrollados específicamente se pueden hacer varias mejoras, como determinar la cantidad de *threads* y bloques óptima de manera automática, modificar el mismo para trabajar con matrices tridimensionales (podría utilizarse para trabajar con imágenes a color) o realizar una interfaz mex para poder llamar al programa desde Matlab u otro software matemático como Octave.

Con respecto a CUDA se recomienda explorar su uso para conseguir implementaciones eficientes de otros algoritmos que exhiban paralelismo o puedan ser expresados de manera paralela. En el SDK de NVIDIA puede encontrarse una gran cantidad de ejemplos de aplicaciones desarrolladas para esta arquitectura. Además, podría incluso probarse las implementaciones en tarjetas más poderosas como la Tesla M2090, la más poderosa en el mercado actualmente, que cuenta con 512 CUDA cores [26].

También sería interesante explorar alternativas como OpenCL para tener la capacidad de portar las implementaciones a tarjetas de video de otros fabricantes y otras arquitecturas. En general se recomienda considerar el uso de tarjetas de video para aplicaciones de propósito general cuando se necesite mejorar las prestaciones de algún programa específico.

Sería también apropiado desarrollar algoritmos que presenten un nivel de paralelismo mayor que el de convolución bidimensional para determinar si incluso en esos casos las aplicaciones optimizadas para CPUs multinúcleo exceden las prestaciones de CUDA.

BIBLIOGRAFÍA

[1] Chapman, B.; Jost, G. & Ruud Van der Pas. Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press, 2007.

[2] Fujimoto, N. Dense Matrix-Vector Multiplication on the CUDA Architecture. Parallel Processing Letters, Vol. 18(4), pp. 511-530, 2008.

[3] Jähne, B. Digital Image Processing. Springer, 2005.

[4] Karimi, K.; Dickson, N. G. & Hamze, F. A Performance Comparison of CUDA and OpenCL. CoRR, Vol. abs/1005.2581, 2010.

[5] Kirk, D. B. & Hwu, W. W. Programming Massively Parallel Processors:A Hands-on Approach. Morgan Kaufmann, 2010.

[6] NVIDIA. CUDA C Best Practices Guide. Version 3.2. NVIDIA, 2010.

[7] NVIDIA. Compute Visual Profiler: User Guide. Version 3.2. NVIDIA, 2010.

[8] NVIDIA. CUDA programming guide. Version 3.2. NVIDIA, 2007.

[9] Patterson, D. A. & Hennessy, J. L. Computer Organization and Design: The Hardware/Software Interface. Morgan Kaufmann, 2005.

[10] Rauber, T. & Rünger, G. Parallel Programming for Multicore and Cluster Systems. Springer, 2010.

[11] Sutter, H. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal, Vol. 30(3), March 2005.

[12] Sutter, H. & Larus, J. Software and the concurrency revolution. ACM Queue, Vol. 3(7), pp. 54-62, 2005.

[13] Wolfe, M. Understanding the CUDA Data Parallel Threading Model: A Primer. The Portland Group Technical News, 2010.

[14] Gonzales, R. C & Woods, R. E. Digital Image Processing 2nd Edition. Prentice Hall. 2002

[15] The Khronos Group. SONY and NVIDIA Become Khronos Promoting Members. Khronos Press Releases. 2005. Tomado de <http://www.khronos.org/news/press/releases/sony-and-nvidia-become-khronos-promoting-members>

[16] The OpenMP API Specification for Parallel Programming. Recurso accesado el 22 de Junio, 2011. <http://openmp.org>

[17] The Open Standard For Parallel Programming of Heterogeneous Systems. Recurso accesado el 22 de Junio, 2011. <http://www.khronos.org/opencl/>

[18] OpenGL: Open Graphics Library. Recurso accesado el 22 de Junio, 2011. <http://www.opengl.org>

[19] BrookGPU. Recurso accesado el 22 de Junio, 2011. <http://graphics.stanford.edu/projects/brookgpu/>

[20] Cg Toolkit. Recurso accesado el 22 de Junio, 2011. <http://developer.nvidia.com/cg-toolkit>

[21] CUDA Computing SDK. Recurso accesado el 22 de Junio, 2011 <http://developer.nvidia.com/gpu-computing-sdk>

[22] Intel Integrated Performance Primitives. Recurso accesado el 22 de Junio, 2011. <http://software.intel.com/en-us/articles/intel-ipp/>

[23] Zeiler, Matthew. IPP Convolution Toolbox V.1.0.0. Abril 4, 2010. New York University. Recurso accesado el 14 de Junio, 2011. <http://www.matthewzeiler.com/software/>

[24] Li, S. Z. & Jain, A. K. (Eds.). Handbook of Face Recognition. Springer, 2004

[25] Michael Haller, M. B. & Rauber, B. T. Emerging Technologies of Augmented Reality: Interfaces and Design. Idea Group, 2007

[26] Tesla M-Class GPU Computing Modules: Fastest parallel Processors for Accelerating Science. Recurso accesado el 20 de Julio, 2011. <http://www.nvidia.com/docs/IO/105880/DS_Tesla-M2090_LR.pdf>