INSTITUTO POLITÉCNICO NACIONAL

CENTRO DE INVESTIGACIÓN Y DESARROLLO DE TECNOLOGÍA DIGITAL

CITEDI
Centro de Investigación y Desarrollo de Tecnología Digital

"PROCESAMIENTO DE IMÁGENES EN UN PROCESADOR GRAFICO"

TESINA

QUE PARA OBTENER LA

ESPECIALIDAD EN SISTEMAS INMERSOS

PRESENTA:

ALMA ROSA LÓPEZ ESCOBEDO

BAJO LA DIRECCIÓN DE:

DR. JUAN JOSÉ TAPIA ARMENTA

JULIO 2010 TIJUANA, B. C., MÉXICO

SIP-14

INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

ACTA DE REVISIÓN DE TESINA

En la Ciudad de Tijuana, B.C. siendo las 12:00 horas del día 1 del mes de julio del 2010 se reunieron los miembros de la Comisión Revisora de Tesina designada por el Colegio de Profesores de Estudios de Posgrado e Investigación de CITEDI para examinar la tesina de especialidad titulada:

PROCESAMIENTO DE IMÁGENES EN UN PROCESADOR GRÁFICO.

Presentada por el alumno:

LÓPEZ	ESCOBEDO	ALMA ROSA
Apellido paterno	materno	nombre(s)

Con registro: B0 9 1 8 8 5

aspirante de:

ESPECIALIDAD EN SISTEMAS INMERSOS

Después de intercambiar opiniones los miembros de la Comisión manifestaron ***SU APROBACIÓN DE LA TESINA***, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

LA COMISIÓN REVISORA

Director de tesina

DR. JUAN JOSE TAPIA ARMENTA

S. E. P.
INSTITUTO POLITECNICO NACIONAL
CENTRO DE INVESTIGACION Y DESARROLLO
DE TECNOLOGIA DIGITAL
DIRECCION

DR. ROBERTO HERRERA CHARLES — M. C. ISAURA GONZÁLEZ RUBIO ACOSTA

EL PRESIDENTE DEL COLEGIO

DR. LUIS ARTURO GONZÁLEZ HERNÁNDEZ

INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

CARTA CESIÓN DE DERECHOS

En la Ciudad de Tijuana, Baja California. el día 1 del mes Julio del año 2010, el (la) que suscribe Alma Rosa López Escobedo

Alumno (a) del Programa TESINA DE ESPECIALIDAD EN SISTEMAS INMERSOS, con número de registro B091885 , adscrito al CENTRO DE INVESTIGACIÓN Y DESARROLLO DE TECNOLOGÍA DIGITAL, manifiesta que es autor (a) intelectual del presente trabajo de Tesina, bajo la dirección de Dr. Juan José Tapia Armenta

Y cede los derechos del trabajo intitulado Procesamiento de imágenes en un procesador gráfico

Al Instituto Politécnico Nacional para su difusión, con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y/o director del trabajo, Este puede ser obtenido escribiendo a la siguiente dirección: Av. Del Parque 1310, Mesa de Otay, Tijuana, Baja California, México CP 22510. Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

Alma Rosa López Escobedo

Nombre y firma

Procesamiento de imágenes en un procesador gráfico

Resumen

Este trabajo presenta de manera general la forma en que se procesan imágenes en una unidad de procesamiento grafico; para ello se aplican diversas funciones implementadas bajo la plataforma CUDA (Arquitectura de dispositivo de cómputo unificado).

Actualmente existe una gran variedad de computadoras potentes, capaces de ejecutar diferentes programas y actividades simultáneamente. Gracias a este tipo de tecnología se han desarrollado diseños gráficos con una mejor calidad.

La demanda de diseños gráficos se ha incrementado, dicha necesidad implica un intercambio de información cada vez más rápido y masivo entre los equipos de cómputo.

El propósito principal de las funciones que se desarrollarán es tener un procesamiento más eficiente, una trasmisión más rápida y reconstrucción de imágenes por medios electrónicos.

El objetivo principal de este trabajo es emplear la plataforma de programación CUDA para desarrollar e implementar dichas funciones. Esta arquitectura de programación está diseñada para crear aplicaciones que de forma transparente, escalen su paralelismo para poder incrementar el número de núcleos computacionales. Para ello, CUDA utiliza tres puntos claves: la jerarquía de grupos de hilos, la memoria compartida y las barreras de sincronización.

Para el desarrollo de la investigación se utiliza el filtro Sobel, éste es un operador diferencial discreto que calcula una aproximación al gradiente de la función de intensidad de una imagen y permite realizar cambios a la misma; entre ellos la detección de bordes, la resolución que se requiera al momento de procesarla, entre otras.

Para la programación del Filtro de Sobel en CUDA, es necesario aplicar diferentes funciones que permitan implementar su funcionamiento de este, y así observar el comportamiento de una imagen presentada.

Palabras claves: CUDA, procesador gráfico, procesamiento de imagen, GPGPU.

Image processing in a graphic processor

Abstract

This paper presents a general way how images are processed in a graphics processing unit, to do that applies various functions implemented under the CUDA platform.

Currently there are a variety of powerful computers capable of running different programs and activities together. Thanks to this technology were developed with better graphics quality.

The demand for graphic design is higher; this need involves an exchange of information increasingly rapid and massive between the computer equipment.

The main purpose of the functions is to develop improved processing, better image transmission and reconstruction electronically.

The main objective of this work is to use CUDA programming platform to develop these functions. This programming model is designed to create applications that transparently scale their parallelism to increase the number of computational cores. To this end, CUDA uses three key points, which are the hierarchy of thread groups, shared memory and synchronization barriers.

For the development of research Sobel filter is used, this is a discrete differential operator that computes an approximation to the gradient of the intensity function of an image, can make changes to it, among them the edge detection, the resolution the time required to process, among others.

To program the Sobel Filter in CUDA, it is necessary to apply different functions to enable the implementation of this operation, and observe the behavior of a displayed image.

Keywords: CUDA, graphic processor, image processing, GPGPU.

Agradecimientos

El trabajo es sagrado, pero puede ser aún más si es bien compartido. Gracias a las personas que les pertenece parte de este trabajo.

Al Dr. Juan José Tapia Armenta mi director y amigo por haber aceptado ayudarme a cumplir con este objetivo, pero sobre todo por su paciencia y apoyo de principio a fin.

A los miembros de mi comité: Dr. Roberto Herrera Charles y ala M.C. Isaura González Rubio Acosta, por sus valiosas observaciones y contribuciones a este trabajo.

A mis profesores, compañeros y amigos por todo lo que aprendí para la profesión, el trabajo y la vida.

Al Centro de Investigación y Desarrollo de Tecnología Digital.

Al Instituto Politécnico Nacional.

Al Consejo Nacional de Ciencia y Tecnología.

Tijuana, Baja California, México

CONTENIDO

Lista de tablas

Lista de figuras

Lista de símbolos y acrónimos

	Español	*Inglés*
PELS	Resolución de conflicto arbitrario	Arbitrary Conflict resolution
CPU	Unidad de procesamiento central	Central Processing Unit
GPU	Unidad de procesamiento gráfico	Graphics Processing Unit
CUDA	Arquitectura de dispositivo de computo Unificado	Compute Unified Device Architecture
PBSM	Memoria compartida por bloque	Per Block Shared Memory
PC	Computadora personal	Personal Computer
MISD	Múltiples instrucciones un solo dato	Multiple Instruction Single Data
SMT	Multi-hebras simultánea	Simultaneously MutiThreading
SSE	Conjunto de instrucciones del tipo SIMD	Type instruction set SIMD
SIMD	Instrucción, múltiples datos	Single Instruction, Multiple Data
ALU	Unidad lógica aritmética	Arithmetic Logic Unit
PCI	Interconexión de componentes periféricos	Peripheral Component Interconnect
RAM	Memoria de acceso aleatorio	Random Access Memory
SDK	Kit de desarrollo de software	Software Development Kit
GLUT	Utilidad de herramientas OpenGL	OPenGL Utility Toolkit
AGP	Puerto de gráficos acelerados	Accelerated Graphics Port

1 Introducción

Las tarjetas gráficas han evolucionado a partir de dispositivos sencillos realizando tareas complicadas; así mismo, los coprocesadores han servido de apoyo a la CPU de varias maneras tales como aplicaciones gráficas, como juegos en 3D computacionalmente exigentes y realistas, que requieren un gran número de operaciones complejas para poder realizar actualizaciones en la imagen. Actualmente las tarjetas graficas contienen Unidades de Procesamiento Gráfico (GPU, del ingles Graphics Processing Unit) programables, que están optimizados para hacer frente a esta carga de trabajo en forma paralela. En términos de rendimiento, la GPU ha superado a los procesadores de propósito general actuales por un amplio margen. Por tanto, hay un gran esfuerzo en muchas comunidades de investigación para utilizar las capacidades de cómputo de las GPU's incluso con fines que no están exclusivamente relacionados con los gráficos de la computadora, incluyendo ciencias de la vida, simulación mecánica, minería de datos y procesamiento de imágenes [4].

Esta área de investigación utiliza las unidades de procesamiento grafico de propósito general, las cuales aprovechan las características específicas de una GPU para realizar tareas donde se requiera un alto nivel de procesamiento.

El procesamiento de imágenes es una de las áreas donde se desarrollan un conjunto de técnicas que se le realizan a una imagen para perfeccionar las características que la forman. Dentro de estas técnicas podemos encontrar el proceso de rasterización, operaciones de fragmentos, entre otras.

Estas técnicas demandan un gran número de operaciones al momento de ser aplicadas, por tal motivo requieren de un dispositivo que sea capaz de procesar datos de manera simultánea.

Para el desarrollo del proyecto se está trabajando bajo la plataforma CUDA, la cual permite procesar datos de manera paralela en una tarjeta gráfica y en la cual se realizaran funciones para el procesamiento de imágenes.

1.1 Objetivos de la investigación

Los objetivos del trabajo se describen a continuación.

1.1.1 Objetivo general

Desarrollar funciones que permitan manipular imágenes mediante programación multi- hilos en un procesador gráfico de una tarjeta de video.

1.1.2 Objetivos específicos

- Realizar un estudio del procesamiento de imágenes.
- Conocer detalladamente el procesador gráfico.
- Aplicar funciones a una imagen y analizarla en un procesador gráfico.

1.1.3 Organización de la tesina

La estructura de este trabajo está organizado de la siguiente manera: En el primer capítulo se presenta una introducción a las tarjetas gráficas así como sus características principales, posteriormente se describen los objetivos generales y específicos de la investigación. El segundo capítulo está orientado al procesamiento de imágenes, y su respectivo esquema gráfico en la generación de imágenes sobre una GPU. El tercer capítulo presenta una descripción detallada sobre los procesadores gráficos. Una descripción de la plataforma CUDA se presenta en el cuarto capítulo. Los resultados de este trabajo se describen en el quinto capítulo. Las conclusiones y trabajo futuro se exponen en el sexto capítulo

2 Procesamiento de imágenes

El tratamiento digital de imágenes contempla principalmente el procesamiento y el análisis de imágenes. El procesamiento está referido a la realización de transformaciones y a la restauración y mejoramiento de las imágenes. El análisis consiste en la extracción de propiedades y características de las imágenes, así como en la clasificación, identificación y reconocimiento de patrones [12].

2.1 Definición de imagen digital

De manera general una imagen digital puede considerarse como una matriz cuyos índices de fila y columna identifican un punto de la imagen y el valor del correspondiente elemento de la matriz indica el color de ese punto. Los elementos de una distribución digital de este tipo se denominan elementos de la imagen o más comúnmente pixeles. Las imágenes digitales se pueden dividir en tres tipos:

- *Monocromáticas*: son imágenes blanco y negro y también imágenes en escala de gris.
- *Colormapped* (paleteada): cada pixel está representado por un número llamado índice que se toma para el color real del pixel desde una tabla llamada paleta.
- *Truecolor:* Estas imágenes son las de más alta calidad y las de mayor ocupación de memoria, cada píxel contiene la información de color completa, usualmente expresada como la intensidad de los componentes de color rojo, verde o azul (RGB) [12].

2.2 Ampliación de la imagen

Para realizar la ampliación de una imagen se utiliza una interpolación después de pasar la imagen del dominio espacial al dominio de la frecuencia mediante la transformada discreta de Fourier. Para hacerlo más eficiente computacionalmente se aplica la transformada rápida de Fourier, para lo cual se requiere que los lados de la imagen tengan un número de pixeles que sea potencia de dos. A continuación se presenta el procedimiento para duplicar cada lado de la imagen, que da como consecuencia que la imagen crezca por un factor de cuatro.

1. Transformar la imagen **A** al dominio de la frecuencia (imagen o matriz **B**).

2. Dividir la imagen transformada (imagen o matriz **B**) en cuatro partes iguales, tal como se muestra en la figura 1.

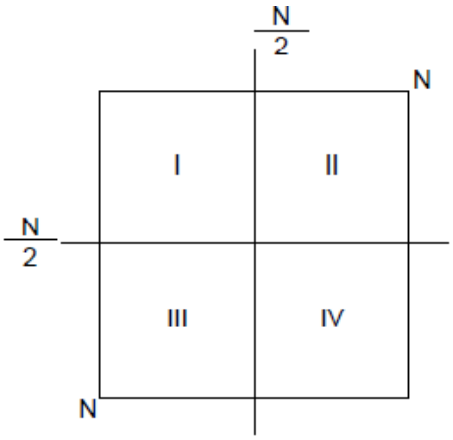


Figura 1: División en cuatro partes iguales de la imagen trasformada (**B**).

3. Insertar *N* ceros a cada renglón de la matriz **B**, como se muestra en la figura 2 (b); enseguida insertar *N* ceros a cada columna, tal como se muestra en la figura 2 (c) para formar una matriz aumentada, **B'**.

4. El siguiente paso es calcular la transformada inversa de Fourier a la matriz aumentada **B'** para obtener una matriz **A'** aumentada, de dimensiones $2N$ x $2N$.

5. Dividir cada elemento de la matriz **X'** entre 64. La relación es $4^{(2n-1)}$, donde *n* es el número de veces que se amplifica la imagen, en este caso $n = 2$.

El procedimiento anterior amplifica la imagen original por un factor de 2 (el área original se cuadruplica). Si se requiere un factor de amplificación diferente de dos, por ejemplo un factor α, deberán agregarse a **B** (α-1) ceros [8].

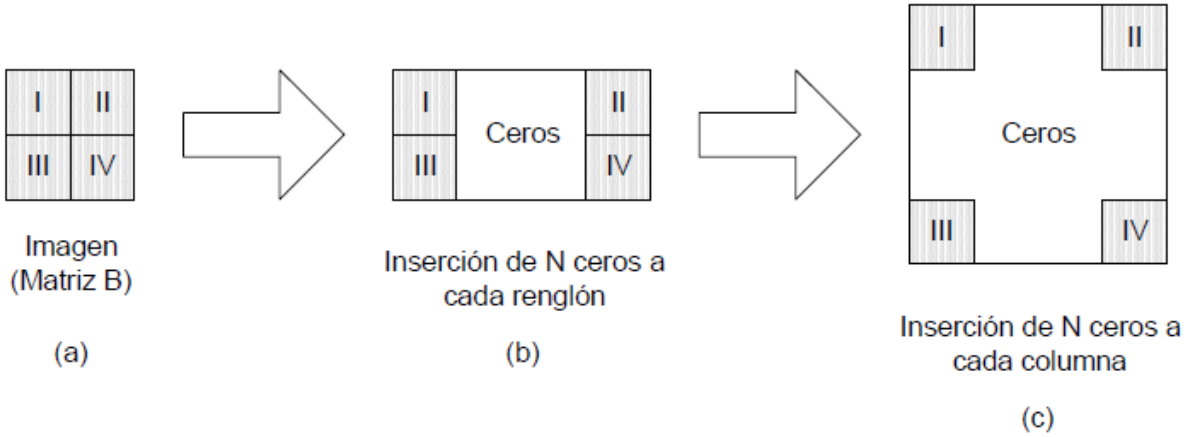


Figura 2: Inserción de ceros para obtener la matriz **B** aumentada (**B'**).

2.3 Resolución de la imagen

La resolución de una imagen indica cuánto detalle puede observarse en ella. El término es comúnmente utilizado en relación a imágenes de fotografía digital, pero también se utiliza para describir la limpieza de una imagen de fotografía convencional (o fotografía química). Tener mayor resolución se traduce en obtener una imagen con más detalle o calidad visual. Para las imágenes digitales almacenadas como mapa de bits, la convención es describir la resolución de la imagen con dos números enteros, donde el primero es la cantidad de columnas de píxeles (cuántos píxeles tiene la imagen a lo ancho) y el segundo es la cantidad de filas de píxeles (cuántos píxeles tiene la imagen a lo alto). El procesamiento digital de imágenes de color se está convirtiendo en una tecnología de base para futuros productos de consumo.
A diferencia de las soluciones del pasado, donde las imágenes de consumo fueron de todo tipo de color dependiendo de la fotografía tradicional, y de esta manera diversas fuentes de imagen, fueron incluyendo medios fotográficos a través de vídeo digital o cámaras, de forma sintética. Cabe mencionar que si la imagen aparece como granular se le da el nombre de pinceleada ó pixelosa [6].
La convención que le sigue en popularidad es describir el número total de píxeles en la imagen (usualmente expresado como la cantidad de megapíxeles), que puede ser calculado multiplicando la cantidad de columnas de píxeles por la cantidad de filas de píxeles. En la figura 3 se presenta una ilustración sobre cómo se vería la misma imagen en diferentes resoluciones.

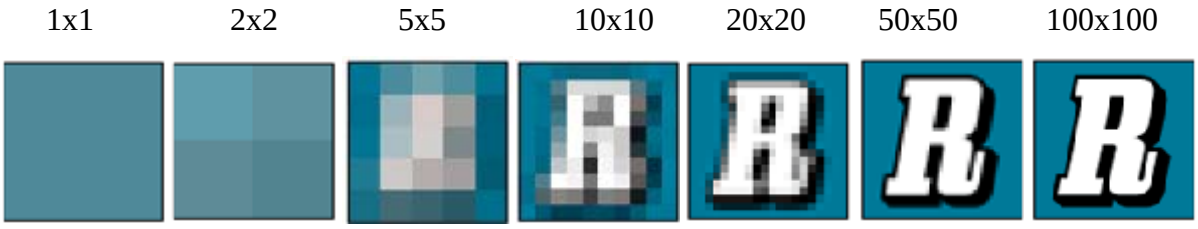


Figura 3: Diferentes resoluciones de la imagen.

2.4 Textura de la imagen

La síntesis de textura es el proceso de crear una imagen digital grande de una imagen digital pequeña mediante la aplicación de un algoritmo. No se trata de un engrosamiento por cambio de la resolución sino de una imitación del contenido de la imagen.

El uso más importante de la síntesis de textura es en la elaboración de imágenes digitales. Aquí es usado para llenar huecos o para engrosar una imagen. Una textura es un patrón de los valores de los pixeles sobre una región pequeña; pero puede ser puesta sobre un objeto o superficie además de que se produce en todo el objeto o sobre una parte seleccionada, esto se aprecia en la figura 4.

Figura 4: Textura de una hoja.

2.5 Calidad de la imagen

Los factores en la estimación de la calidad de la imagen son la resolución, el color y el tono, así como el aspecto general. Para debatir más ampliamente acerca del sistema de medición de la calidad de las imágenes consultar [6].

2.5.1 Resolución

La resolución es el factor clave en la determinación de la calidad de imagen de materiales de texto y otras representaciones con base a los bordes. Para los materiales gráficos, especialmente imágenes de tono continuo, la profundidad de bits, la representación del color, y el rango dinámico, se combinan con la resolución para determinar la calidad. Los atributos de la resolución que deben ser inspeccionados son la legibilidad, integridad, oscuridad, contraste,

nitidez, y uniformidad. La medición y evaluación de cada trazo y detalle son útiles para valorar la calidad de la imagen.

2.5.2 Color y tono

Para las imágenes en color, a escala de grises y para algunas imágenes monocromáticas, la reproducción del color y del tono son indicadores significativos de la calidad, complementando el "detalle" proporcionado por la resolución. El objetivo que se persigue al hacer la valoración de la apariencia del color y del tono es determinar en qué medida una imagen transmite la misma apariencia al tiempo que el color y el tono varían respecto del documento original (o del intermedio utilizado). La valoración del tono y del color puede ser altamente subjetiva y cambiante de acuerdo con el entorno de visualización y las características de los monitores y de las impresoras.

A continuación se presenta una serie de ilustraciones que demuestran los efectos del color y el tono sobre la calidad de la imagen.

Figura 5: Demostración de rango dinámico limitado.

En la figura 5 se muestra el concepto de rango dinámico en donde en la imagen de la izquierda se puede observar que ésta más limitada con respecto a la figura de la derecha, en especial el detalle en cuanto a las sombras y los toques de luz.

Figura 6: Demostración del concepto de profundidad de bits.

En la figura 6 se puede apreciar cuando una imagen se reduce de 24 bits (izquierda) a una de 8 bits (derecha), la reducción del color puede tener como resultado artefactos de cuantificación.

(a) (b)

Figura 7: Demostración del brillo y contraste.

El brillo y el contraste son parámetros importantes en el procesamiento de imágenes pero cabe mencionar que el exceso y la falta de brillo afectan la calidad de la imagen. En la figura 7 se puede apreciar el concepto de brillo y contraste, en donde la imagen (b) con respecto a la imagen (a) posee un alto brillo y contraste. Se observa que el exceso de brillo en la imagen (b) reduce la calidad de imagen

2.5.3 Evaluación general

La calidad de la imagen es acumulativa, afectada por una variedad de factores individuales tales como el rendimiento del sistema de captura, resolución, rango dinámico y precisión del color.

La evaluación final debería realizarse sobre la imagen en general, apreciando todos los factores individuales que contribuyen a la calidad.

2.6 Filtro de Sobel

El operador filtro o filtro de Sobel se emplea para acentuar los bordes en una imagen, su funcionamiento radica en utilizar una vecindad de 3x3 pixeles. La implementación de este filtro está dada por:

$$G(x, y)=\sqrt{6x^2 + 6y^2}\ , \tag{1}$$

donde

$$\begin{aligned} Gx&=(A_2 + 2A_3 + A_4) - (A_0 + 2A_7 + A_6) \\ Gy&=(A_0 + 2A_1 + A_2) - (A_6 + 2A_5 + A_4). \end{aligned} \tag{2}$$

Siendo A_i los pixeles de la ventana, en las posiciones:

$$\begin{Bmatrix} A_1 & A_1 & A_2 \\ A_7 & g(x,y) & A_3 \\ A_6 & A_5 & A_4 \end{Bmatrix} \tag{3}$$

2.7 Detección de bordes basado en el operador sobel

El filtro de Sobel está basado en un operador gradiente el cual se puede calcular de diferentes maneras, sin embargo los operadores de Sobel tienen la ventaja de proporcionar tanto una diferenciación como un efecto de suavizado, razón por la cual fue seleccionado para realizar las pruebas [1]. Estos operadores se muestran en la Figura 8.

1	2	1
0	0	0
-1	-2	-1

(a)

1	0	-1
2	0	-2
1	0	-1

(b)

Figura 8: Mascara sobel para calcular(a) gradiante G*x* y (b) gradiante G*y*.

Para calcular dos componentes gradiente de Gx y Gy, en las orientaciones horizontales y verticales respectivamente, se aplican las siguientes relaciones (4) y (5).

$$Gx=[f(i-1,j-1)+2f(i-1,j)+f(i-1,j)+f(i-1,j+1)]-[f(i+1,j-1)+2f(i+1,j)+f(i+1,j+1)] \quad (4)$$

$$Gy=[f(i-1,j-1)+2f(i,j-1)+f(i+1,j-1)]-[f(i-1,j+1)+2f(i,j+1)+f(i+1,j+1)] \quad (5)$$

La magnitud del gradiente es generalmente calculada como:

$$G\,[f(x,y)]=\sqrt{G_x^2+G_y^2} \quad (6)$$

Sin embargo las otras dos formulas de las ecuaciones (4) y (5) también se pueden utilizar para calcular la magnitud del gradiente, la dirección del gradiente se calcula a través de la formula de la ecuación (6). El resultado de una imagen generada por el borde del operador sobel se muestra en la figura 9.

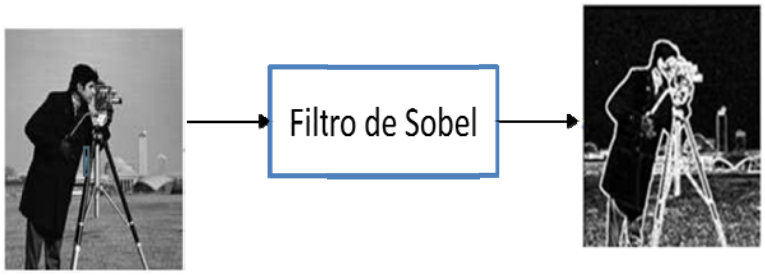


Figura 9: Filtro Sobel aplicado a una imagen.

La magnitud y la dirección del gradiente son de importancia en la detección de bordes, porque brindan información valiosa que se puede utilizar para aceptar o rechazar un borde que no posea determinadas características [2]. Las etapas para la detección de bordes en una imagen se muestran en la figura 10.

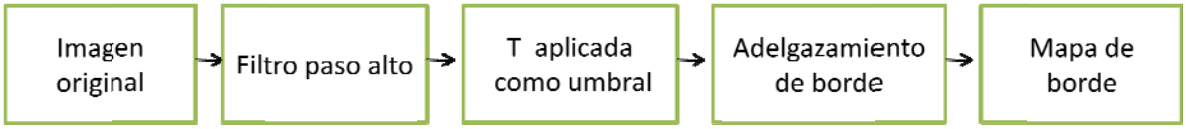


Figura 10: Proceso de detección de bordes.

3 Procesadores gráficos

Hoy en día los procesadores gráficos han evolucionado hasta el punto en donde muchas de las aplicaciones industriales se ejecutan en diversos niveles; es decir, su rendimiento es muy superior a los que se ofrecían al momento de ejecutarse en sistemas multi-núcleo. Gracias a la tecnología, las arquitecturas informáticas se han desarrollado de tal manera que se pueden establecer sistemas híbridos con GPUs compuestas por núcleos de procesamiento paralelo que trabajan en colaboración con las CPUs multi-núcleo desarrollando diversas actividades de manera simultánea.

3.1 Antecedentes

Tal y como hoy en día se conocen, las tarjetas gráficas componen una pieza fundamental dentro del mundo de los gráficos. Fueron creadas para ofrecer rapidez en sus cálculos, capaces de ofrecer una apariencia de realismo para las aplicaciones gráficas como películas de animación simuladores o videojuegos, liberando de dichas tareas al procesador central [5].

Una tarjeta gráfica es una tarjeta de expansión para una computadora encargada de procesar los datos que vienen de la CPU y transformarlos en información comprensible y representable en un dispositivo de salida tales, como un monitor o un televisor. Está formada por múltiples componentes, entre ellos se encuentran;

- La GPU: es el componente clave de la tarjeta.
- Memoria de video: Es una memoria rápida que ha evolucionado mucho durante los últimos años. Permite a la tarjeta manejar toda la información visual que le envía la CPU del sistema. Su tamaño varía con el modelo de tarjeta. La memoria empleada en los últimos años está basada en la tecnología DDR, destacando DDR2, GDDR3, GDDR4 y GDDR5.
- Ramdac: Es un conversor de señal digital a analógica, se encarga de transformar las señales digitales producidas en la computadora en una señal analógica que sea interpretable por el monitor.
- Disipador: Conocido como dispositivo pasivo compuesto de material conductor del calor, que extrae éste de la tarjeta. Su eficiencia va en función de la estructura y la superficie total por lo que son bastante voluminosos.

- Ventilador: corresponde a un dispositivo activo que aleja el calor almacenado de la tarjeta al mover el aire cercano. Es menos eficiente que un disipador y produce ruido al tener partes móviles.
- Alimentación: hace unos años la alimentación eléctrica de las tarjetas gráficas no había supuesto un gran problema, sin embargo la tendencia actual de las nuevas tarjetas es consumir más energía. Aunque las fuentes de alimentación son cada día más potentes, el cuello de botella se encuentra en el puerto PCIe. Por este motivo, las tarjetas gráficas con un consumo superior al que puede suministrar el PCIe incluyen un conector (PCIe *power connector*) que permite una conexión directa entre la fuente de alimentación y la tarjeta sin tener que pasar por la placa base.

De tal manera que con la aparición de novedosos video juegos con grandes efectos y con mayores requerimientos de hardware, se pensó en la distribución del trabajo entre el GPU y el CPU para una mejor optimización en los recursos de hardware. No era suficiente tener trabajando al 100% el GPU o el CPU, si no que se podía distribuir el trabajo en estas dos unidades de procesamiento y así pudieran trabajar en forma paralela para obtener un mejor rendimiento en los videojuegos o aplicaciones 3D. En la actualidad están apareciendo nuevas tecnologías en cuanto a la aplicación de GPU´s y CPU´s; en donde se pueden tener en una computadora estos dos elementos trabajando en paralelo para obtener un máximo desempeño en el hardware y la posibilidad de más y mejores efectos visuales.

3.2 Arquitectura de una GPU

Una GPU tiene un diseño conocido como arquitectura *Streaming*. Dentro de esta se encuentran tres formas de paralelismo [13]:

- Paralelismo de tareas: En un entorno multi-hilo, el procesador puede soportar la conexión de varios clientes y asignar a cada uno un subconjunto del banco de registros para que su flujo de datos atraviesen el cauce segmentado de la forma disjunta.

  En el caso de las GPUs este tipo de paralelismo se da en casos puntuales cuando se puede trabajar de forma desacoplada y simultánea con la CPU o cuando se tienen varias GPUs conectadas realizando diferentes tareas.

- Paralelismo a nivel instrucción: Este mecanismo consiste en romper el flujo secuencial de instrucciones para simultanear la ejecución de varias en el mismo procesador. Existen diferentes estrategias para lograrlo, descritas a continuación:

  Segmentación: Las instrucciones se dividen en etapas de igual duración y utilizan una unidad funcional diferente cada una. Suelen consistir en búsqueda de la instrucción, decodificación, lectura de operandos, ejecución y escritura de resultados. Así mientras una instrucción está en la fase de decodificación, la siguiente está en la de búsqueda.

  Superescalaridad: Consiste en la replicación de unidades funcionales de manera que se puedan ejecutar varias instrucciones a la vez en el mismo chip. Es compatible con la segmentación.

  Supersegmentación: Es aquél que presenta varios niveles de segmentación. Así algunas de las etapas que mencionamos para los procesadores segmentados, es dividida a su vez en dos o más etapas de manera que se permite que haya dos instrucciones a la vez dentro de la misma etapa y unidad funcional sin tener que replicar esta.

- Paralelismo de datos: es un paradigma de la programación concurrente que consiste en subdividir el conjunto de datos de entrada a un programa, de manera que a cada procesador le corresponda un subconjunto de esos datos. Cada procesador efectuará la misma secuencia de operaciones que los otros procesadores sobre su subconjunto de datos asignado. En resumen: se distribuyen los datos y se replican las tareas.

  Así como también es un paradigma suficientemente adecuado para operaciones sobre vectores y matrices, dado que muchas de ellas consisten en aplicar la misma operación sobre cada uno de sus elementos.

El diseño de las CPU realiza procesos para reducir el número de accesos a la memoria principal en área del chip, ya que está principalmente cubierta por una jerarquía de memoria, el objetivo de dicha jerarquía es conseguir una memoria de gran velocidad, que provoque limitación en el uso de un mayor número de transistores como unidades aritméticos lógicas (ALU) y unidades de punto flotante (FPU).

Sin embargo, en las GPUs se tiene un modelo circulante que hace que la arquitectura reduzca considerablemente el gasto energético necesario para que unas etapas y otras accedan a la información y como consecuencia tener un porcentaje dedicado a la computación aritmética.

3.2.1 Tipos de procesadores gráficos

Las principales compañías desarrolladoras de GPUs son NVIDIA y ATI (adquirida por AMD en el 2006).

- **ATI:** ATI Technologies Inc. Es una de las mayores empresas de hardware que diseña procesadores gráficos y tarjetas de video. Su mercado acapara todo tipo de productos para el procesamiento gráfico y multimedia, tanto para computadoras personales, como para dispositivos portátiles, videoconsolas, teléfonos móviles y televisión digital. Fue fundada el 20 de agosto de 1985.
- **NVIDIA:** es una empresa multinacional especializada en el desarrollo de unidades de procesamiento gráfico y tecnologías de circuitos integrados para estaciones de trabajo, ordenadores personales y dispositivos móviles. Con sede en Santa Clara, California, la compañía se ha convertido en uno de los principales proveedores de circuitos integrados como unidades de procesamiento grafico y conjuntos de chips usados en tarjetas de gráficos en videoconsolas y placas base de computadora personal.

3.3 Funcionamiento del procesador gráfico

Una GPU está altamente segmentada y posee gran cantidad de unidades funcionales. Gracias a la segmentación consigue una alta capacidad de procesamiento al aplicar ingeniosamente el paralelismo, que es la tendencia actual en la arquitectura de procesadores para computadoras personales. Las CPUs modernas incorporan varios núcleos de procesamiento; por ejemplo, el intel Core Duo o Intel Quad. Por otro lado una GPU común (Geforce 7800 GTX o Geforce 8800 GTX), poseen 32 y 128 unidades de procesamiento respectivamente. El procesador de una tarjeta gráfica es diferente al de una computadora normal. Mientras que los núcleos de la CPU pueden trabajar de manera independiente, los núcleos de una GPU dependen unos de otros. Sin embargo, queda claro la existencia del paralelismo en las arquitecturas actuales, característica esencial del procesador gráfico.

Estos procesadores se dividen principalmente en dos: aquellos especializados en procesar vértices y pixeles, Se establece al pixel y al vértice como los principales componentes que maneja la GPU.

3.3.1 Flujo de datos

Cauce gráfico y su relación con la arquitectura del GPU

Para procesar una imagen en una GPU se lleva a cabo una serie de pasos, este esquema se representa en la figura 11.

Las operaciones del renderizado se aplican siempre en un mismo orden, inician por las que se encuentran relacionadas con el atributo de posición del vértice y finalizan con las que son más cercanas al atributo del color del pixel [3].

Figura 11: Esquema gráfico en la generación de imágenes sobre una GPU.

La CPU toma un lista precalculada con los vértices de todos los polígonos de la escena y los envía a la GPU. Así inicialmente la GPU recibe la información de la CPU en forma de vértices lo que evita enviar toda la información de la imagen por lo que el tráfico en el bus (PCI o AGP) es mucho menor en la mayoría de las veces enviando únicamente los vértices, que todas las características de la escena.

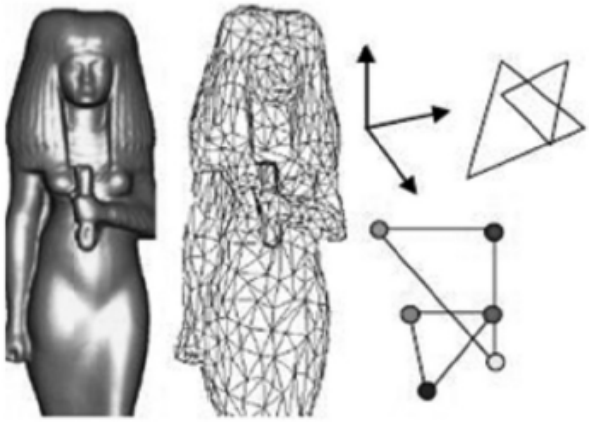


Figura 12: Forma de triangulación para enviarle los vértices a la GPU.

Procesador de vértice: El tratamiento que reciben los vértices se realiza en el procesador de vértices. Un vértice es la esquina de una primitiva donde se unen los bordes que lo delimitan.

Un triangulo tiene tres vértices y los objetos que conocemos hoy en día pueden formarse por miles de triángulos, así con solo definir tres vértices se puede generar todos lo pixeles que lo constituyen.

Un vértice lleva información sobre sus coordenadas, color, peso, textura, tamaño y con ellos se realizan trasformaciones, como la traslación o la rotación de las figuras. El procesador cambia la posición de los vértices, lo que afecta a la posición en donde se dibuja el objeto final, y es así como una GPU tiene varios procesadores de vértices [3].

Rasterizado: Tras la manipulación de los vértices se define la parte de ellos que se va a visualizar (técnica llamada *clipping*) y se eliminan los triángulos cuya cara no está orientada hacia la cámara (técnica denominada *culling*). De esta manera, no se trabaja con objetos o partes de un objeto en que en la escena final no van a ser visibles. Esto sucede en la etapa de rasterización, donde cada polígono es convertido a una serie de fragmentos del tamaño de un pixel. Este paso se encarga de convertir cada punto, línea o polígono 3D en una matriz 2D de puntos, donde se guarda la información acerca del color y profundidad entre otras, realizando una interpolación de cada uno de los puntos que lo forman.

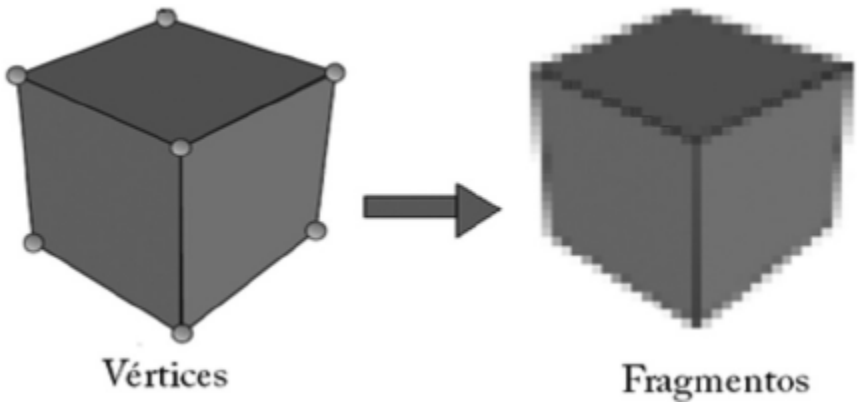


Figura 13: Proceso de rasterizado.

Procesador de fragmentos: El resultado de la etapa de rasterización es lo que le llega al procesador de fragmentos. Un fragmento es un conjunto de datos que se produce tras la rasterización de las primitivas (triángulos, puntos, cuadrados, entre otras). En este hardware se realizan las trasformaciones referentes a los fragmentos tales como la aplicación de texturas o transformaciones sobre los fragmentos que finalmente constituyen el pixel a visualizar.

Hoy en día las GPUs cuentan con varias unidades de procesamiento, disponiendo así de más procesadores de vértices (en la mayoría de los casos). Esto se debe a que en el cauce aumenta el volumen de datos a medida que se recorre el proceso de fragmentos, la cual se encuentra al final del mismo y se necesitan unidades en este punto para que haya un equilibrio y no se produzca un cuello de botella. Dentro del proceso también se pasan múltiples tests para eliminar aquellos objetos que queden ocultos por otros y así no queden dentro del rango de la cámara.

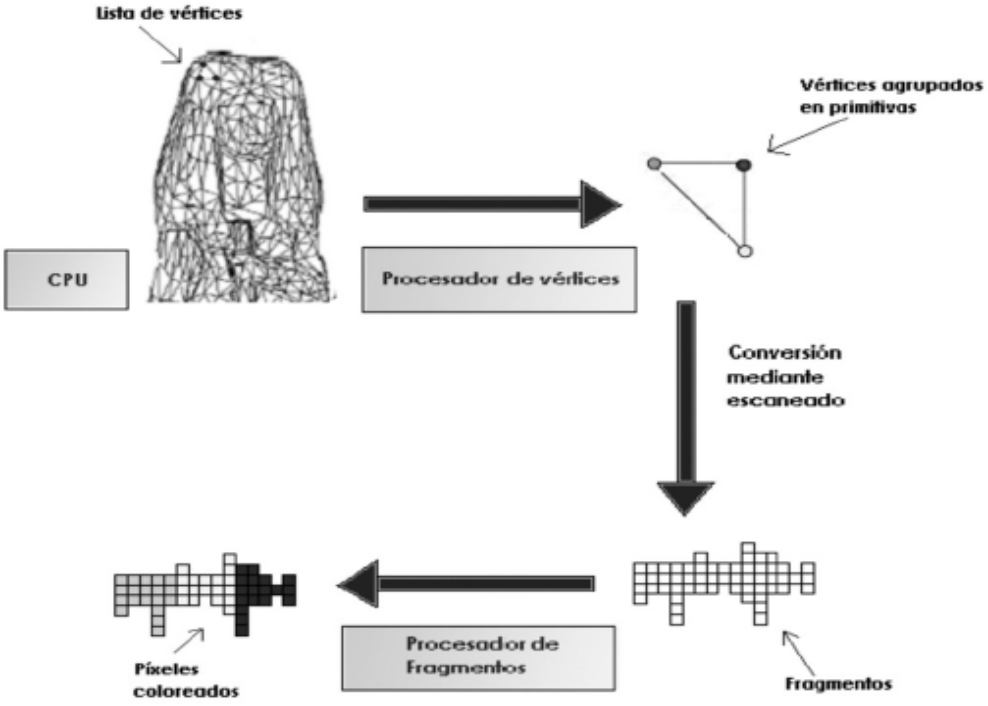
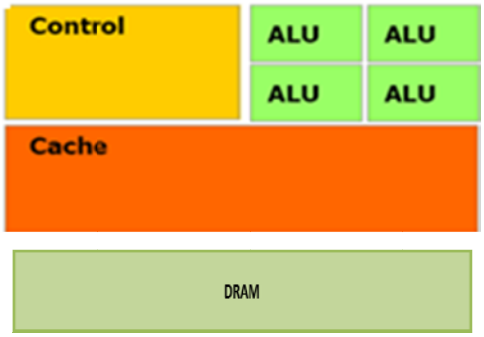


Figura 14: Resumen del cauce gráfico en la GPU.

3.3.2 Diferencia entre el CPU y GPU

La principal diferencia que existe entre la CPU y la GPU es que ésta última cuenta con un mayor número de transistores que se dedican al procesamiento de datos, y por lo tanto hace que las operaciones se ejecuten de una manera más rápida [7]. Esta diferencia se puede apreciar fácilmente en la figura15.

CPU: Unidad de Procesamiento Central



GPU: Unidad de Procesamiento Gráfico

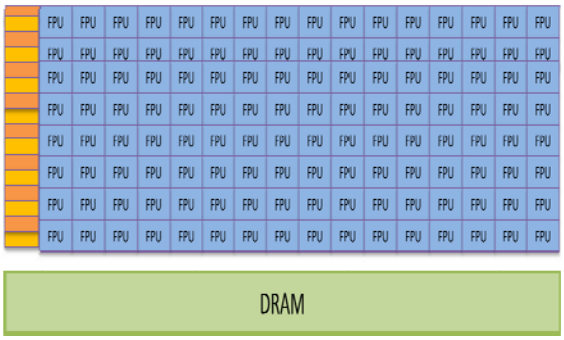


Figura 15: Diferencia en el diseño entre la CPU y GPU.

Otra diferencia se muestra en la figura 16, donde se puede observar que el incremento de la velocidad de las CPU´s es más bajo que el incremento de las GPU´s; los CPU´s han incrementado en promedio 2 veces su poder de procesamiento cada 18 meses.

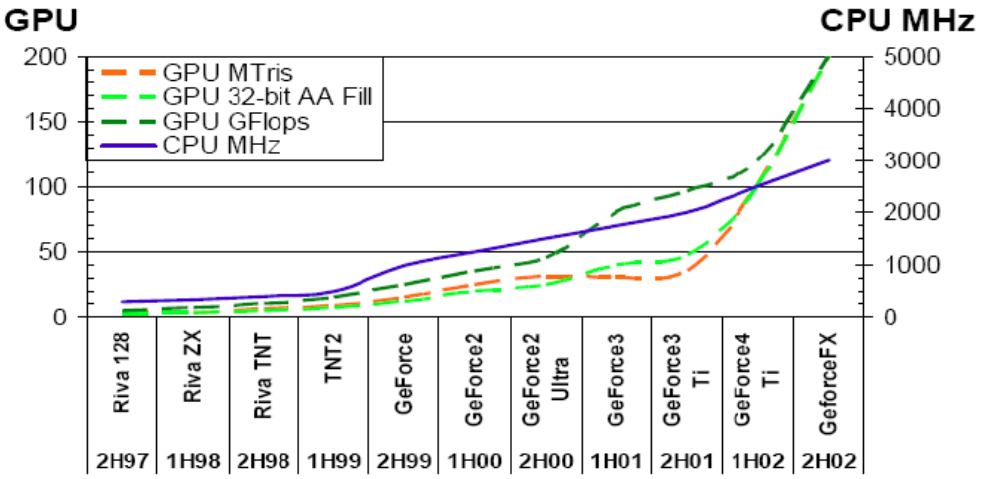


Figura 16: Comparación entre el incremento de velocidad de los CPU´s y los GPU´s.

3.4 Procesador GT220

Debido a la gran capacidad que tienen los procesadores gráficos para realizar tareas en paralelo, ha surgido gran interés para aplicarlos en áreas como la medicina, modelado de clima, investigación científica, entre otras; en donde el poder de procesamiento que se requiere para resolver problemas es mucho más grande que en aquellos donde se procesen menos datos. Por lo anterior, ha surgido lo que se conoce como unidades de procesamiento gráfico de propósito

general (GPGPU); estas además de tener la función de procesar imágenes pueden ser utilizadas para resolver problemas de propósito general.

NVIDIA lanzó una plataforma en el 2007 llamada CUDA, en la cual los desarrolladores pueden programar las GPUs para realizar cualquier tipo de aplicaciones. Esta herramienta es compatible con cualquier tarjeta gráfica de NVIDIA, a partir de la serie 8 en adelante. Cada una de estas tarjetas gráficas cuenta con una GPU (y algunas cuentan con dos) que la hacen con altas capacidades para procesar datos en paralelo.

Las tarjetas gráficas de la serie 200 fueron lanzadas en el año 2009, dentro de ellas se encuentra la GT 220, utilizada en el desarrollo de investigación, esta se muestra en la figura17.

Figura 17: Tarjeta gráfica NVIDIA GeForce GT 220.

3.4.1 Características

La tabla 1 presenta las características de la tarjeta grafica GeForce GT 220.

Tabla 1: Especificaciones de la tarjeta gráfica GeForce GT 220.

Memoria Global	1024 MB
Número de multiprocesadores	6
FPU´s por multiprocesador	8
Números de cores	48
Memoria constante	64 KB
Memoria compartida por bloque	16 KB
Número de registros disponibles por bloque	16384
Tamaño Warp	32 Hilos
Número máximo de hilos por bloque	
Máximo tamaño de cada dimensión de un bloque	512x512x64
Máximo tamaño de cada dimensión de una malla (grid)	65535x65535x1

Memoria Global: Es la memoria total con la que cuenta el procesador gráfico y es compartida por todos los hilos de la malla, independientemente del bloque en que se encuentre [11].

Número de multiprocesadores: Es la cantidad de unidades de procesamiento con las que cuenta cada unidad de punto flotante (FPU); y es aquí en donde se ejecutan los bloques de la malla.

FPU’s por multiprocesador: Son las unidades encargadas de realizar las operaciones aritméticas y lógicas del GPU; en estas unidades son ejecutadas los hilos de cada *warp*.

Números de cores: Es el número total de unidades de procesamiento con las que cuenta una unidad de procesamiento gráfico, en cada una de ellas se ejecutan las llamadas a las funciones kernels.

Memoria Constante: Es un espacio de memoria de solo lectura, que puede ser accesible por todos los hilos al igual que con la memoria global.

Memoria compartida por bloque: Es el total de memoria que comparte cada uno de los bloques de hilos y únicamente pueden ser accedidos por los hilos que se encuentran dentro del bloque.
Número de registros disponibles por bloque: Es la cantidad de registros que se encuentran dentro de los bloques de hilos.

Tamaño *Warp*: Los hilos de bloque se agrupan en *Warps* para su ejecución y un *warp* equivale a 32 hilos y sus hilos se ejecutan en un procesador.

Número máximo de hilos por bloque: Es la cantidad total de hilos en un bloque, cada hilo puede:

- Leer/Escribir registros locales al hilo.
- Leer/Escribir memoria local al hilo.
- Leer/Escribir memoria compartida por todo el bloque.
- Leer/Escribir memoria global compartida por toda la malla (grid).
- Solo lectura de las memorias constantes y de textura.

Tamaño máximo de cada dimensión de un bloque: es el número total de hilos que hay en un bloque.

Tamaño máximo de cada dimensión de una malla: Es el número total de bloques contenidos en una malla.

3.4.2 Diseño del procesador GT 220

La capacidad de procesamiento de este GPU se basa principalmente en la forma en que está diseñado, debido a que este cuenta con 8 unidades de punto flotante y cada uno de ellos con 6 multiprocesadores, que hacen un total de 48 unidades de procesamiento. Este diseño se presenta en la figura18.

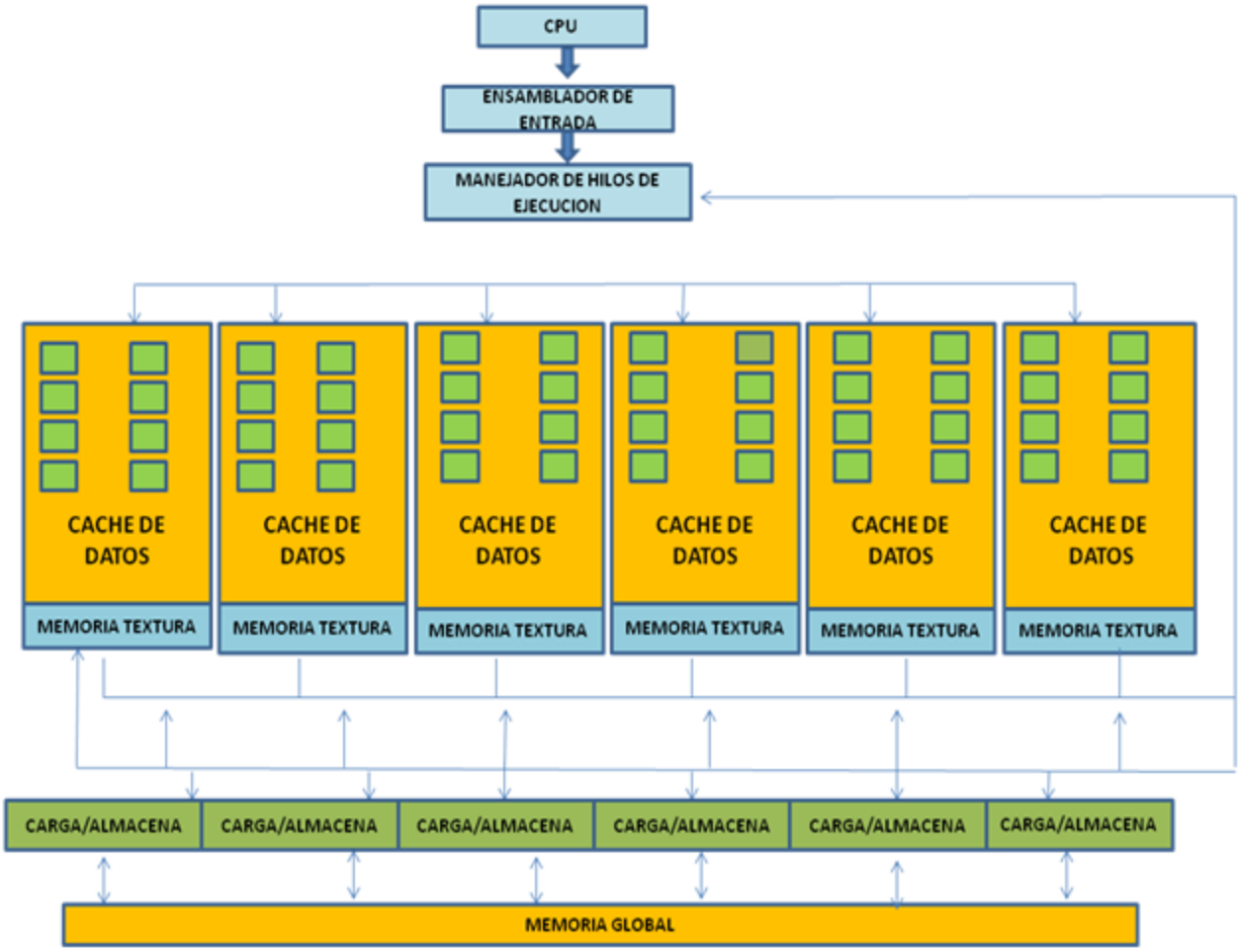


Figura 18: Diseño del GPU GT 220.

4 Plataforma CUDA

CUDA es la arquitectura de computación paralela de NVIDIA que permite un aumento importante en el rendimiento de los cálculos al aprovechar la potencia de la GPU [9]. Es también un modelo de programación de propósito general que consiste en desarrollar un conjunto de herramientas desarrolladas por NVIDIA para poder programar las tarjetas graficas que fabrican. Dentro de este conjunto de herramientas está el compilador CUDA, manejador para la tarjeta, un kit de desarrollo con proyectos, entre otros. En CUDA se utiliza el lenguaje de programación C para desarrollar aplicaciones con la GPU.

4.1 Flujo de datos

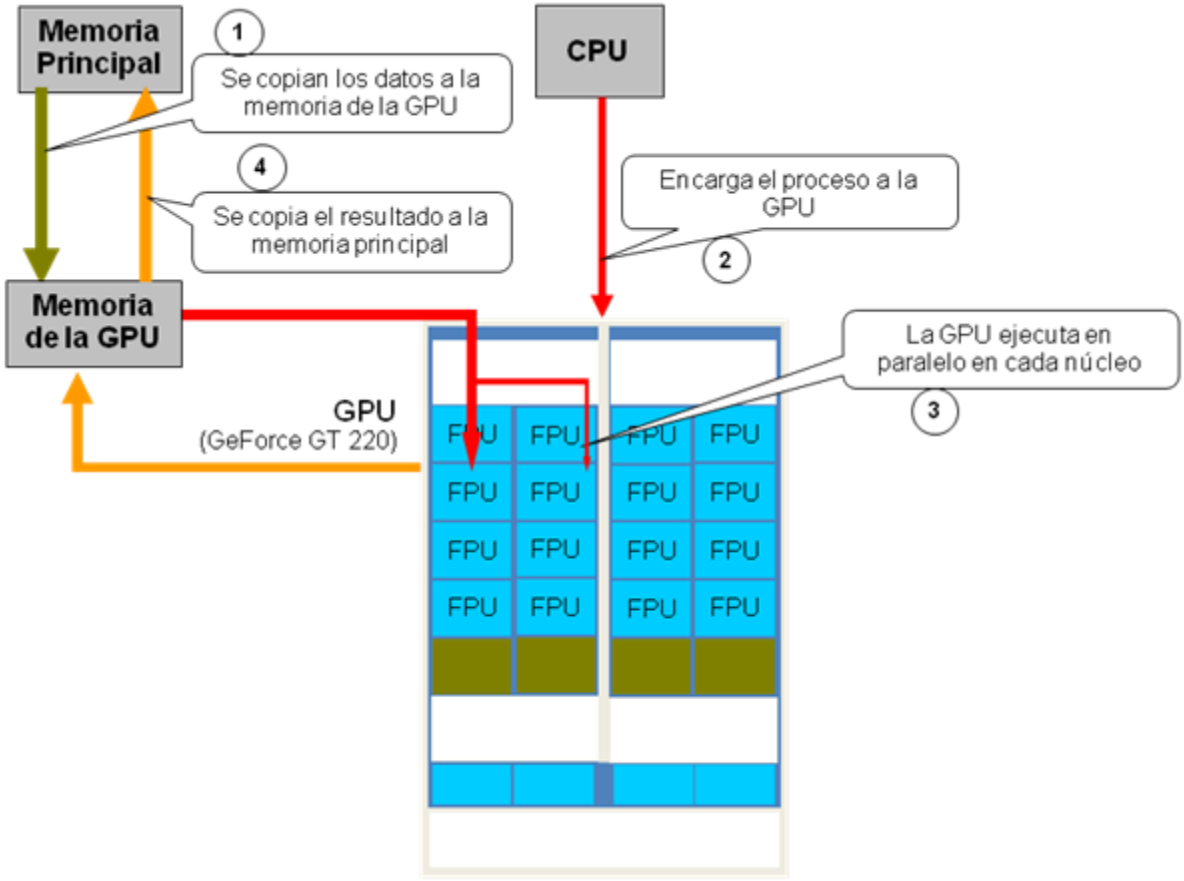


Figura 19: Flujo de procesamiento en CUDA.

En la figura 19 se presenta el flujo de procesamiento en CUDA y se describe a continuación:

1) La primera etapa consiste en copiar los datos de la memoria principal (memoria del host) a la memoria de la GPU (memoria del dispositivo), esto con la finalidad de que los datos que se ejecuten en paralelo en cada núcleo estén presentes en la RAM de la GPU y poder así realizar el procedimiento más rápido.
2) La segunda etapa consiste en autorizar el proceso a la GPU. Los datos que requieran un proceso secuencial se ejecutarán en el CPU (*host*), mientras que aquellos que requieran un proceso paralelo el CPU encargará esta función a la GPU.
3) El siguiente paso es sin duda el más importante, ya que es aquí en donde la unidad de procesamiento gráfico ejecuta en paralelo los datos que estén presentes en la memoria, esto gracias a la arquitectura que posee una GPU y a sus múltiples unidades de procesamiento.
4) Finalmente, el resultado obtenido del procesamiento paralelo se copia de la memoria de la GPU hacia la memoria principal, para que sean procesados por el CPU.

4.2 Modelo CUDA

La GPU y la CPU son tratados como dispositivos separados con su propio espacio de memoria, esto se puede observar en la figura 20.

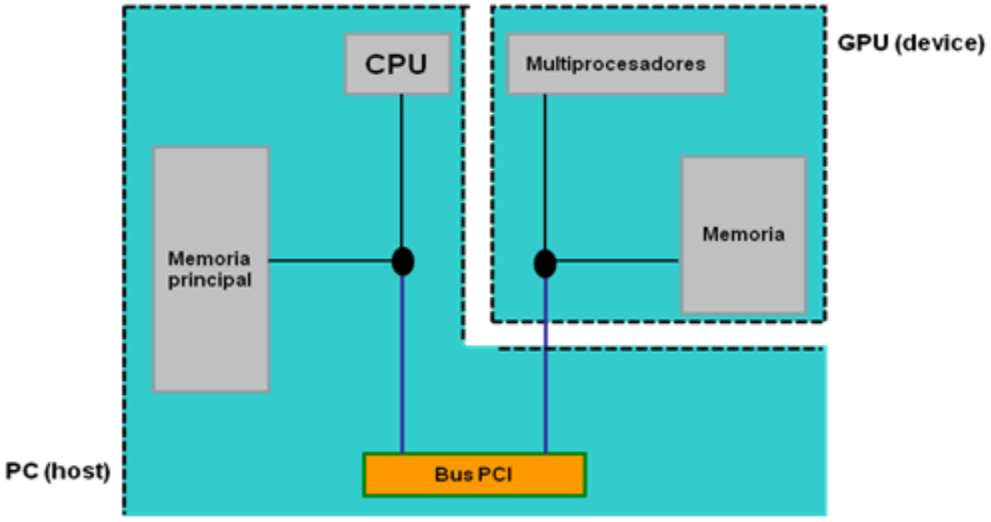


Figura 20: Modelo CUDA.

De la figura anterior se deriva lo siguiente:

- ✓ La GPU funciona como coprocesador de la CPU.
- ✓ Tiene su propia RAM.
- ✓ Ejecuta multitud de hilos en paralelo.

Las principales características del modelo CUDA son las siguientes:

- ✓ La GPU no puede acceder directamente a la memoria principal.
- ✓ La CPU no puede acceder directamente a la memoria de la GPU.
- ✓ La transferencia de datos necesita ser explícita.
- ✓ No es posible ejecutar *printf* en la GPU.

Las abstracciones que se manejan en CUDA implican la jerarquía de hilos y la jerarquía de memoria; esto debido a la forma en que está organizada la GPU. Además, otra de las características que hacen diferente a este modelo es que proporciona escalabilidad; es decir, un programa CUDA puede ejecutarse sobre cualquier número de procesadores, porque no es necesario recompilar el código.

4.2.1 Jerarquía de hilos

Una función que se ejecuta en un procesador gráfico que soporta CUDA se le llama *kernel.* Al ejecutar un *kernel* se crea una malla (*grid*) de bloques, que a su vez cada uno contiene varios hilos (*threads*). En la figura 21 se observa la jerarquía de hilos en CUDA:

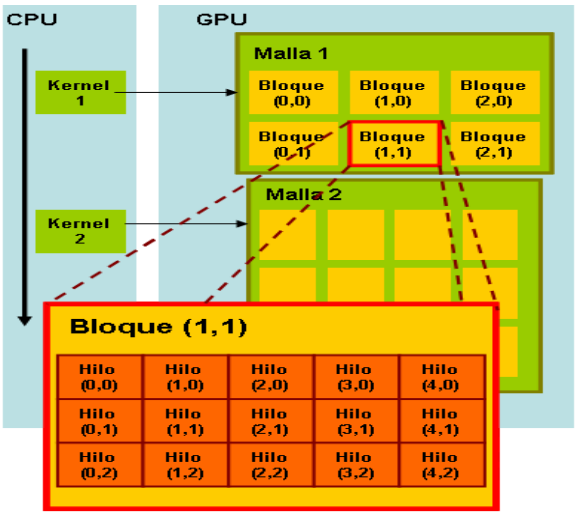


Figura 21: Jerarquía de hilos en CUDA.

Todos los hilos comparten la memoria de datos. Un bloque de hilos es un lote de hilos que pueden cooperar entre ellos: sincronizándose o compartiendo datos mediante una memoria de muy baja latencia [10].

Cada bloque y cada hilo están identificados dentro de cada malla, por las variables predefinidas *blockIdx* y *threadIdx* que pueden ser: *blockIdx* 1D o 2D y, *threadIdx* 1D, 2D o 3D. De esta forma se simplifica el direccionamiento al procesar datos multidimensionales: imágenes, matrices, entre otros.

4.2.2 Espacio de memoria

Los hilos en CUDA pueden acceder a distintas memorias; unas compartidas y otras no:

- ✓ Cada hilo dispone de una memoria privada a la que solo él puede acceder.

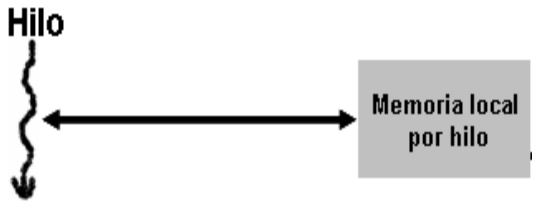


Figura 22: Memoria local por un hilo.

- ✓ Cada bloque posee un espacio de memoria que es compartida por los hilos del bloque. El tiempo de vida de esta memoria es igual a la del propio bloque.

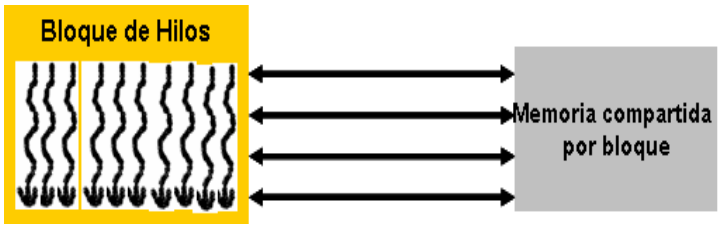


Figura 23: Memoria compartida por bloque.

- ✓ Existe una memoria global a la que todos los hilos pueden acceder, independientemente del bloque en el que se ejecuten, e incluso independientemente de la malla sobre el que se esté corriendo.

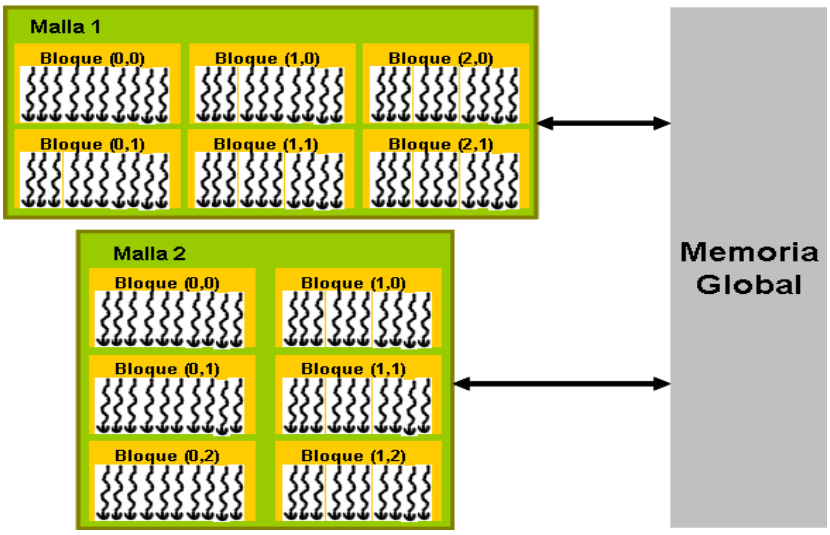


Figura 24: Memoria global.

Además, existen otros dos espacios de memoria de solo lectura que pueden ser accesibles por todos los hilos al igual que la memoria global. Estas son la zona de memoria constante y la de textura.

4.2.3 Gestión de la memoria

Para gestionar la memoria, CUDA dispone de las siguientes funciones:

a) Reservar y liberar espacio en la memoria de la GPU

✓ *cudaMalloc()*

Obtiene espacio en la memoria global del dispositivo. Requiere dos parámetros: dirección del puntero y tamaño a reservar.

✓ *cudaFree()*

Libera el objeto que se le envía como parámetro.

b) Copia de memoria entre dispositivo

✓ *cudaMemcpy()*

Esta función se utiliza para transferir datos entre las memorias. Requiere de cuatro parámetros: puntero al origen, puntero al destino, tamaño de los datos a copiar y tipo de transferencia; que pueden ser cuatro:

- Host a host
- Host a dispositivo

- Dispositivo a host
- Dispositivo a dispositivo

4.2.4 Sincronización

Cuando se ejecuta un *kernel* en CUDA, los programadores no se encargan del procedimiento de sincronización de los hilos esta función es realizada por CUDA; ya que facilita el uso de la función *void __syncthreads*(). La ejecución de un hilo de un bloque no continuará hasta que todos los hilos del mismo bloque hayan llegado a esa llamada de sincronización.

4.3 Instalación y configuración

El proceso de instalación y configuración de CUDA se lleva a cabo a través de varios pasos que a continuación se mencionan:

a) Se instala el driver para la tarjeta gráfica. Esto se debe de realizar como *root* y se requiere detener el demonio del modo gráfico (servicio *gdm*)

b) Debido a que algunas aplicaciones necesitan la biblioteca de funciones de openGL, se requiere instalar aquellas que sean compatibles con el *driver* correspondiente.

c) Habiendo instalado el *driver*, se reinicia la computadora y se puede verificar que se realizó correctamente corriendo la instrucción "*Nvidia-settings*", la cual muestra como resultado la ventana que se muestra en la figura 25.

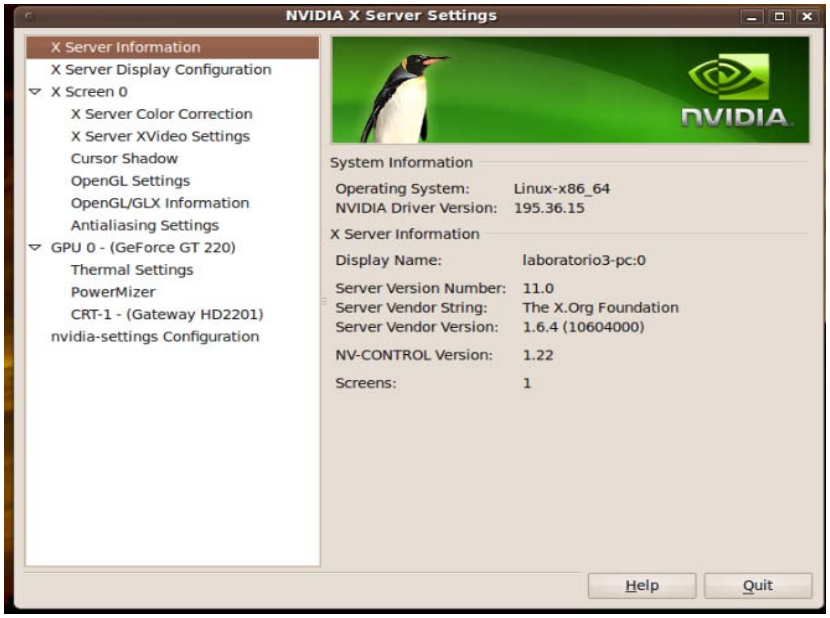


Figura 25: Información de la configuración del driver de *NVIDIA.*

d) Posteriormente se instala el *toolkit*, que al igual que el *driver* se realiza como *root.* Aquí es en donde se instala el compilador de NVIDIA (*nvcc*), algunos archivos bin, los archivos de ayuda, entre otros.

e) Instalado el *toolkit* se procede realizar lo mismo con el SDK de NVIDIA, que en este caso se instaló la versión 3.0. Para llevar a cabo esto no es necesario ser root, es decir, se puede instalar como usuario normal. Al momento de instalar el SDK se pide la ruta donde se desea instalarlo, por defecto, se instala en la carpeta *home* del usuario; así mismo, se solicita la ruta donde se instalo CUDA (*toolkit*); que normalmente se encuentra en la ruta */usr/local/cuda* del sistema de archivos de linux.

f) Si se tiene una distribución de linux reciente que tiene como compilador gcc (versión 4.4); será necesario instalar la versión 4.3 del gcc debido a que CUDA no trabaja aun con versiones actuales de éste. Después de instalarlo, se deben de realizar los enlaces correspondientes hacia el compilador *nvcc* para que funcione correctamente.

g) Finalmente se instalan algunos paquetes que son necesarios para que *openGL* funcione adecuadamente; entre ellos el paquete GLUT, que es la Utilidad Toolkit de openGL y que consiste en un sistema de ventanas independientes Toolkit para escribir programas en openGL.

Para comprobar el proceso anterior se ejecutan algunos proyectos que se incluyen en el SDK de NVIDIA. Algunos ejemplos de ello se muestran en las figuras 26 y 27.

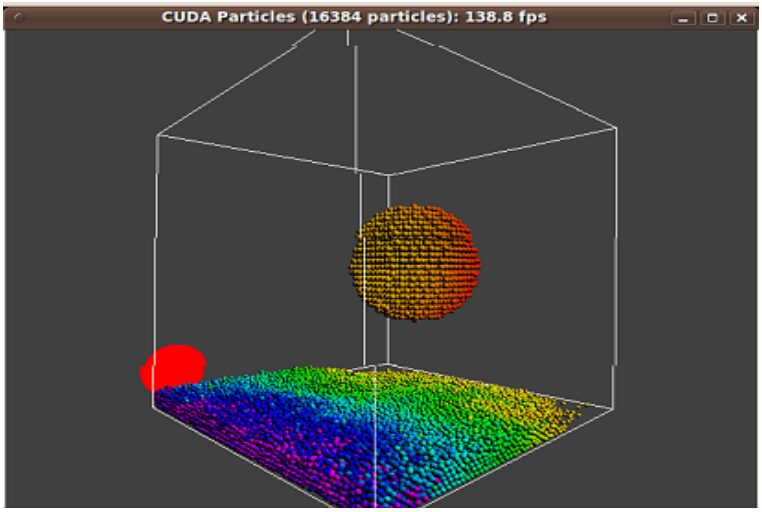


Figura 26: Ejemplo Particles.

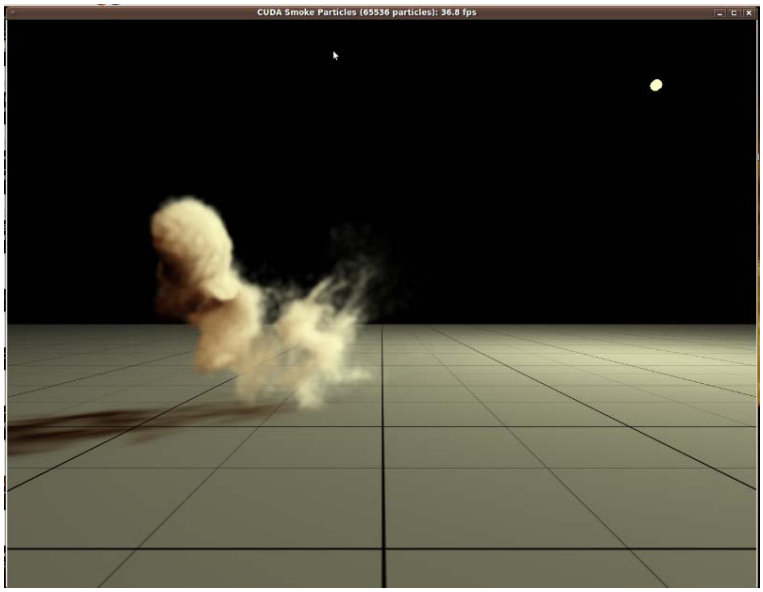


Figura 27: Ejemplo smokeParticles.

4.4 Modelo de programación CUDA

La arquitectura de hardware paralelo CUDA está acompañado por el modelo de programación paralela CUDA que provee un conjunto de abstracciones que permiten expresar los datos en grano fino y grano grueso y paralelismo de tareas. El programador puede elegir un lenguaje de programación de alto nivel para expresar el paralelismo como C, C++, Fortran ó driver API como OpenGL y cómputo DirectX-11. El primer lenguaje que NVIDIA dio soporte, fue para el lenguaje C. Un entorno de C para la herramienta de desarrollo de software CUDA permite programar el GPU usando C con un mínimo de palabras claves o extensiones. El modelo de programación paralela CUDA guía al programador en la partición del problema en sub-problemas de grano grueso que se pueden resolver independientemente en paralelo. El paralelismo en grano fino, y los sub-problemas pueden ser resueltos cooperativamente en paralelo.

4.4.1 Aplicación

Para procesar una imagen en CUDA es necesario realizar las siguientes operaciones:

1.- Escribir el *kernel* que se va a ejecutar en la malla

Este procedimiento consiste en definir la cantidad de pixeles con la que se va a procesar la imagen. Para definir un *kernel* en CUDA se hace de la siguiente manera:
_global_void “nombre_kernel”(parámetros).
Esta función se explica en el apéndice A.

2.- Reservar memoria de la GPU

Este paso consiste en reservar memoria en el procesador gráfico utilizando la función cudaMalloc, de la siguiente manera:

```
cudaMalloc( (void **)&d_result, imWidth*imHeight*sizeof(Pixel));
```

donde:

- `&d_result` es el puntero del espacio de memoria a reservar
- `imWidth*imHeight*sizeof(Pixel)` es el tamaño de memoria reservada

3.-Copiar datos de/hacia la GPU

En esta parte se copian los datos del GPU al CPU. En este caso la función utilizada es:

```
cudaMemcpy(g_CheckRender->imageData(), d_result,
imWidth*imHeight*sizeof(Pixel), cudaMemcpyDeviceToHost);
```

4.-Ejecutar/invocar kernels

Este procedimiento consiste llamar la función SobelShared desde la CPU, con los siguientes parámetros:

```
SobelShared( uchar4 *pSobelOriginal, unsigned short SobelPitch)
```

5.-Compilación y ejecutar

Para compilar el proyecto es necesario posicionarse en la carpeta
NVIDIA_GPU_Computing_SDK/C $
Y ejecutar el comando make.
Después de compilar se generan archivos ejecutables que son guardados en la carpeta C/bin/linux/release/.
El siguiente paso es ubicarnos en la carpeta NVIDIA_GPU_Computing_SDK/C/bin/linux /release y ejecutar el comando; ./SobelFirter

SobelFilter: es el archivo del proyecto con el que se está trabajando.

5 Resultados

Para realizar el procesamiento de imágenes en el procesador gráfico se utilizó el algoritmo Filtro de Sobel, este se escribió en el lenguaje de programación C de CUDA y fue implementado en el sistema operativo Linux. El desarrollo del proyecto se basó en los siguientes recursos.

Tabla 2: Recursos de Hardware y Software

Hardware	Software
✓ Procesador (CPU): Intel Core 2 Quad Q6600 (2.4 GHz) ✓ Memoria RAM: 6 GB ✓ Tarjeta Grafica (GPU): - Nvidia Geforce GT 220	✓ Sistema operativo Linux Ubuntu 9.10 (*Kernel* 2.6.31-20) ✓ Versión gcc: 4.4.1 (Ubuntu) ✓ Versión CUDA: 3.0

Algunas de las funciones que se consideraron importantes para el procesamiento de la imagen fueron las siguientes.

LoadDefaultImage: Esta función carga la imagen a la memoria de la GPU para poder procesarla y luego inicializarla con la función initializeData como se muestra en el apéndice B.

Para poder visualizar la imagen se necesitan los siguientes valores.

```
static int imWidth  = 30;
static int imHeight = 30;
```

Los resultados obtenidos al momento de procesar la imagen a través de la aplicación del Filtro de Sobel fueron:

- Mejor resolución en la imagen

Figura 28: Imagen escala a grises.

En figura 28 se observa la imagen original que es utilizada para aplicarle las operaciones del filtro de sobel; la cual se muestra a escala de grises.

- Aumento de brillo en la imagen

Figura 29: Imagen con intensidad luminosa.

Al presionar las teclas correspondientes del brillo de la imagen (=, -), se puede apreciar el aumento o disminución del brillo. Para esto el programa utiliza la variable global *imageScale*, que se utiliza para fijar la escala del brillo de la imagen. En la figura 29 se muestra la imagen con un aumento de brillo a la imagen.

- Detección de bordes

Figura 30: Detección de bordes.

En la figura 30, se puede apreciar claramente la aplicación del filtro de Sobel; en donde se marcan los bordes de la imagen que están dado por la asignación de los pixeles de la matriz que se muestra a continuación.

```
unsigned char pix00 = LocalBlock[SharedIdx+4*ib+0*SharedPitch+0];
unsigned char pix01 = LocalBlock[SharedIdx+4*ib+0*SharedPitch+1];
unsigned char pix02 = LocalBlock[SharedIdx+4*ib+0*SharedPitch+2];
unsigned char pix10 = LocalBlock[SharedIdx+4*ib+1*SharedPitch+0];
unsigned char pix11 = LocalBlock[SharedIdx+4*ib+1*SharedPitch+1];
unsigned char pix12 = LocalBlock[SharedIdx+4*ib+1*SharedPitch+2];
unsigned char pix20 = LocalBlock[SharedIdx+4*ib+2*SharedPitch+0];
unsigned char pix21 = LocalBlock[SharedIdx+4*ib+2*SharedPitch+1];
unsigned char pix22 = LocalBlock[SharedIdx+4*ib+2*SharedPitch+2];
```

Es importante aclarar que se realizaron pruebas a la imagen con diferentes cantidades de pixeles para observar la importancia que tiene al momento de ser procesada aplicando el operador Sobel, ya que este calcula el gradiente de la intensidad de brillo de cada pixel dando la dirección del mayor incremento posible de negro a blanco.

5.1 Visualización de la imagen a través de openGL

OpenGL es una interfaz software de hardware grafico, es decir, define las funciones que se pueden utilizar en una aplicación para acceder a las prestaciones de un dispositivo grafico.
La librería para utilizar openGL es <GL/glut.h>. Aquí están definidas las funciones para el manejo de gráficos con openGL.

GLUT (*OpenGL Utility Toolkit*) es una biblioteca que permite construir aplicaciones interactivas basadas en eventos en las que el elemento principal es openGL. Esta biblioteca incluye principalmente las siguientes funciones:

glutInit(): Esta función se encarga de la inicialización de GLUT

glutInitDisplayMode(): Con esta función se indican las características que se desean para la visualización con OpenGL. Algunos parámetros de esta función son:

- ✓ GLUT_DOUBLE: Memoria de imágenes doble
- ✓ GLUT_RGBA: Memoria de imagen con componentes rojo, verde, azul y alfa para cada pixel

glutInitWindowSize (): Esta función permite indica el tamaño de la ventana

glutCreateWindow (): Esta función se utiliza para crear la ventana y lleva como parámetro en titulo de la venta

glutDisplayFunc(): Esta función dibuja la ventana para mostrar la imagen

glTexImage2D(): Esta función es para especificar la imagen que va hacer utilizada como textura. Los parámetros de esta función son:

void glTexImage2D(GLenum target, GLint level, GLint components, GLsizei width,GLsizei height,GLint dorder,GLenum format,GLenum type, const GLvoid *pixels).

Las funciones anteriores se muestran en el apéndice B; las cuales se utilizaron para el procesamiento y visualización de la imagen aplicando el filtro de Sobel.

6 Conclusiones y trabajo futuro

En este trabajo se presentó un análisis del procesamiento digital de las imágenes a través del filtro de Sobel, que se basa principalmente en procesar la imagen en un procesador gráfico de la marca NVIDIA, así como también en ir modificando cada valor de los pixeles de acuerdo a la operación que se le vaya aplicando en la entrada de la imagen. Algunas de estas operaciones, utilizan ciertos arreglos de valores arbitrarios para ir modificando la imagen de acuerdo a la operación seleccionada por el usuario. Otras de estas operaciones, utilizan matrices transpuestas para modificar los píxeles de acuerdo a las propiedades geométricas de la imagen para realizar las operaciones. Es importante mencionar que, aplicando distintas operaciones a la imagen de entrada, se pueden obtener diversos resultados, los cuales se aplican en diferentes áreas de la investigación, como por ejemplo en el servicio meteorológico, en el área de medicina entre otras. Esto debido a que algunos de estos resultados son de gran importancia para estas ramas de la ciencia.

Como trabajo futuro se propone aplicar el filtro de Roberts y el de Prewitt ya que a través de ellos se obtendrá una mejor aproximación al momento de calcular los bordes, así como también poder observar la diferencia respecto a un número mayor de pixeles y de esta manera obtener una aproximación de lo que hay alrededor del punto donde se requiera hacer un cálculo, para lo cual este trabajo proporciona soporte para cualquier proceso de aplicación de una imagen.

Referencias

[1] A. Tinru y K.Ray, *Image processing participles y Applications,* Wiley, 2005.

[2] A. Austin, *Handbook of image video processing* , Texas, 1999.

[3] B. Iran, F. Tim, H. Daniel y S. Jeremy, *Brook for GPUs: stream computing on graphics hardware,* Stanford University, 2004.

[4] B. David, y S. Chief, NVIDIA *CUDA Software and GPU parallel Computing Architecture*, 2008.

[5] C. Daniel y T. Philippas, *GPU-Quicksort: A practical Quicksort Algorithm for Graphics Processors,* ACM journal of experimental Algorithmic vol.14,No.1.4.pp.24,July 2009.

[6] C. Johnton, D. Bailey y P. Lyons, *A visual Environment for real-time image processing in hardware(vertiph),* Eurasip journal on embedded systems, volume 2006, pp.1-8,2006.

[7] F. Kayvon y H. Mike, *GPUs a closer look*, Stanford University, April 2008.

[8] J. Pinoli y J. Debayle, *Logarithmic Adaptative Neighborhood image processing (LANIP): Introduction*, Connections to Human Brightness perception and Application Issues, Eurasip journal on Applied signal processing volume 2006, pp. 1-12,2005.

[9] k. Andrew, D. Gregory y M. Sudhakar, *Modeling GPU. CPU Workloads and system*, of electrical and computer Engineering Georgia Institute of Technology Atlanta, march 2010.

[10] N. John, B. Ian y G. Michael, *Scalable parallel programming*, University of Virginia, April 2008.

[11] P. Tosic, *A perspective on the future of massively parallel computing: Fine-Grain vs. Coarse-Grain parallel Models,* University of llinois at Urbana Champaign (UIUC), April 2004.

[12] R. Danijela y G. Axel, *Performance Measure as Feedback variable in image processing*, Eurasip journal on Applied signal processing, volume2006,pp. 1-12, 2005.

[13] W. Greg y T. Christian, *Teaching Parallel Computing: New possibilities School of computing & information systems,* Grand valley state university, October 2009.

Apéndice A. Programa en CUDA para aplicar el filtro Sobel

Programa escrito en lenguaje C

```
__global__ void
SobelShared( uchar4 *pSobelOriginal, unsigned short SobelPitch,
#ifndef FIXED_BLOCKWIDTH
             short BlockWidth, short SharedPitch,
#endif
             short w, short h, float fScale )
{
    short u = 4*blockIdx.x*BlockWidth;
    short v = blockIdx.y*blockDim.y + threadIdx.y;
    short ib;

    int SharedIdx = threadIdx.y * SharedPitch;

    for( ib = threadIdx.x; ib < BlockWidth+2*Radius; ib += blockDim.x
) {
        LocalBlock[SharedIdx+4*ib+0] = tex2D( tex,
            (float) (u+4*ib-Radius+0), (float) (v-Radius) );
        LocalBlock[SharedIdx+4*ib+1] = tex2D( tex,
            (float) (u+4*ib-Radius+1), (float) (v-Radius) );
        LocalBlock[SharedIdx+4*ib+2] = tex2D( tex,
            (float) (u+4*ib-Radius+2), (float) (v-Radius) );
        LocalBlock[SharedIdx+4*ib+3] = tex2D( tex,
            (float) (u+4*ib-Radius+3), (float) (v-Radius) );
    }
    if ( threadIdx.y < Radius*2 ) {
        //
        // copia final* Radio 2 filas de pixeles en compartir
        //
        SharedIdx = (blockDim.y+threadIdx.y) * SharedPitch;
        for ( ib = threadIdx.x; ib < BlockWidth+2*Radius; ib +=
blockDim.x ) {
            LocalBlock[SharedIdx+4*ib+0] = tex2D( tex,
                (float) (u+4*ib-Radius+0), (float) (v+blockDim.y-
Radius) );
            LocalBlock[SharedIdx+4*ib+1] = tex2D( tex,
                (float) (u+4*ib-Radius+1), (float) (v+blockDim.y-
Radius) );
            LocalBlock[SharedIdx+4*ib+2] = tex2D( tex,
                (float) (u+4*ib-Radius+2), (float) (v+blockDim.y-
Radius) );
            LocalBlock[SharedIdx+4*ib+3] = tex2D( tex,
```

```
                (float) (u+4*ib-Radius+3), (float) (v+blockDim.y-
Radius) );
        }
    }

    __syncthreads();

    u >>= 2;    // indice  como uchar4 de aqui
    uchar4 *pSobel = (uchar4 *) (((char *)
pSobelOriginal)+v*SobelPitch);
    SharedIdx = threadIdx.y * SharedPitch;

    for ( ib = threadIdx.x; ib < BlockWidth; ib += blockDim.x ) {

        unsigned char pix00 =
LocalBlock[SharedIdx+4*ib+0*SharedPitch+0];
        unsigned char pix01 =
LocalBlock[SharedIdx+4*ib+0*SharedPitch+1];
        unsigned char pix02 =
LocalBlock[SharedIdx+4*ib+0*SharedPitch+2];
        unsigned char pix10 =
LocalBlock[SharedIdx+4*ib+1*SharedPitch+0];
        unsigned char pix11 =
LocalBlock[SharedIdx+4*ib+1*SharedPitch+1];
        unsigned char pix12 =
LocalBlock[SharedIdx+4*ib+1*SharedPitch+2];
        unsigned char pix20 =
LocalBlock[SharedIdx+4*ib+2*SharedPitch+0];
        unsigned char pix21 =
LocalBlock[SharedIdx+4*ib+2*SharedPitch+1];
        unsigned char pix22 =
LocalBlock[SharedIdx+4*ib+2*SharedPitch+2];

        uchar4 out;

        out.x = ComputeSobel(pix00, pix01, pix02,
                             pix10, pix11, pix12,
                             pix20, pix21, pix22, fScale );

        pix00 = LocalBlock[SharedIdx+4*ib+0*SharedPitch+3];
        pix10 = LocalBlock[SharedIdx+4*ib+1*SharedPitch+3];
        pix20 = LocalBlock[SharedIdx+4*ib+2*SharedPitch+3];
        out.y = ComputeSobel(pix01, pix02, pix00,
                             pix11, pix12, pix10,
                             pix21, pix22, pix20, fScale );

        pix01 = LocalBlock[SharedIdx+4*ib+0*SharedPitch+4];
        pix11 = LocalBlock[SharedIdx+4*ib+1*SharedPitch+4];
        pix21 = LocalBlock[SharedIdx+4*ib+2*SharedPitch+4];
        out.z = ComputeSobel( pix02, pix00, pix01,
                              pix12, pix10, pix11,
                              pix22, pix20, pix21, fScale );
```

```
        pix02 = LocalBlock[SharedIdx+4*ib+0*SharedPitch+5];
        pix12 = LocalBlock[SharedIdx+4*ib+1*SharedPitch+5];
        pix22 = LocalBlock[SharedIdx+4*ib+2*SharedPitch+5];
        out.w = ComputeSobel( pix00, pix01, pix02,
                              pix10, pix11, pix12,
                              pix20, pix21, pix22, fScale );
        if ( u+ib < w/4 && v < h ) {
            pSobel[u+ib] = out;
        }
    }

    __syncthreads();
}
```

Apéndice B. Función para configurar de la textura de la imagen

```
void initializeData(char *file) {
    GLint bsize;
    unsigned int w, h;
    size_t file_length= strlen(file);  //Se obtiene la longitud de la
imagen que se le pasa

    if (!strcmp(&file[file_length-3], "pgm")) {
        if (cutLoadPGMub(file, &pixels, &w, &h) != CUTTrue) {
            printf("Failed to load image file: %s\n", file); //Si hay
un error al momento de cargar la imagen
            exit(-1);
        }
        g_Bpp = 1;
    } else if (!strcmp(&file[file_length-3], "ppm")) {
        if (cutLoadPPM4ub(file, &pixels, &w, &h) != CUTTrue) {
            printf("Failed to load image file: %s\n", file);     //Si
hay un error al momento de cargar la imagen
            exit(-1);
        }
        g_Bpp = 4;
    } else {
        cudaThreadExit();  //
        exit(-1);
    }
    imWidth = (int)w;  //Obtiene el ancho de la imagen, con la
cantidad de pixeles dados
    imHeight = (int)h; //Obtiene el alto de la imagen, con la cantidad
de pixeles dados
    //imWidth = 15;  //Obtiene el ancho de la imagen
    //imHeight = 15; //Obtiene el alto de la imagen
    printf("\n\ng_Bpp=%u\n",g_Bpp);    //g_Bpp=1
    printf("\n\nimWidth=%d\n",imWidth);
    printf("\n\nimHeight=%d\n",imHeight);
    setupTexture(imWidth, imHeight, pixels, g_Bpp);
    printf("\nbytes=%d\n",g_Bpp * sizeof(Pixel) * imWidth * imHeight);
    memset(pixels, 0x0, g_Bpp * sizeof(Pixel) * imWidth *
imHeight);//Fija los primeros bytes(sizeof(Pixel) * imWidth *
imHeight)
                                                    //del bloque de memoria
apuntados por pixels
    if (!g_bQAReadback) {
        // usa OpenGL Path
        glGenBuffers(1, &pbo_buffer);
        glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo_buffer);
        glBufferData(GL_PIXEL_UNPACK_BUFFER,
                     g_Bpp * sizeof(Pixel) * imWidth * imHeight,
```

```
                        pixels, GL_STREAM_DRAW);

        glGetBufferParameteriv(GL_PIXEL_UNPACK_BUFFER, GL_BUFFER_SIZE,
&bsize);
        if ((GLuint)bsize != (g_Bpp * sizeof(Pixel) * imWidth *
imHeight)) {
            printf("Buffer object (%d) has incorrect size (%d).\n",
(unsigned)pbo_buffer, (unsigned)bsize);
            cudaThreadExit();
            exit(-1);
        }

        glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);

            // registra este objeto buffer  con cuda

cutilSafeCall(cudaGraphicsGLRegisterBuffer(&cuda_pbo_resource,
pbo_buffer, cudaGraphicsMapFlagsWriteDiscard));

        glGenTextures(1, &texid);
        glBindTexture(GL_TEXTURE_2D, texid);
        glTexImage2D(GL_TEXTURE_2D, 0, ((g_Bpp==1) ? GL_LUMINANCE :
GL_BGRA),
                     imWidth, imHeight,  0, GL_LUMINANCE,
GL_UNSIGNED_BYTE, NULL);
        glBindTexture(GL_TEXTURE_2D, 0);

        glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
        glPixelStorei(GL_PACK_ALIGNMENT, 1);
    }
}
```