



UNIVERSIDAD NACIONAL AUTÓNOMA DE  
MÉXICO

---

---

POSGRADO EN CIENCIA E INGENIERÍA DE LA  
COMPUTACIÓN

DETECCIÓN Y RECONOCIMIENTO DE OBJETOS  
UTILIZANDO TÉCNICAS DE VISIÓN EN GPU

T E S I S

QUE PARA OBTENER EL GRADO DE:  
MAESTRO EN INGENIERÍA EN COMPUTACIÓN

PRESENTA:  
JAIME ALAN MÁRQUEZ MONTES

DIRECTORES DE TESIS:  
DR. JESÚS SAVAGE CARMONA  
DR. JOSE DAVID FLORES PEÑALOZA

CIUDAD DE MÉXICO, D. F.

JUNIO, 2015

# **Detección y Reconocimiento de objetos utilizando técnicas de visión en GPU**

por

Jaime Alan Márquez Montes

Tesis presentada para obtener el grado de

Maestro en Ingeniería en Computación

en el

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Ciudad de México, D. F.. Junio, 2015

*Dedicatorias.....*

.....  
.....

## AGRADECIMIENTOS

Agradecimientos.....

.....

.....

## TABLA DE CONTENIDO

<b>1. Introducción</b>	<b>1</b>
1.1. Contexto . . . . .	1
1.2. Problema a resolver . . . . .	2
1.3. Hipótesis . . . . .	2
1.4. Estructura de la tesis . . . . .	3
<b>2. Marco Teórico</b>	<b>4</b>
2.1. Extracción y descripción de características . . . . .	4
2.1.1. Características invariantes a transformaciones afines . . . . .	5
2.2. Cómputo en GPU . . . . .	11
<b>3. Unidad de Procesamiento de Graficos de Proposito General</b>	<b>14</b>
3.1. Breve Historia de las GPU . . . . .	15
3.2. Plataforma y modelo de programación de cómputo paralelo . . . . .	16
3.2.1. Arquitecturas . . . . .	17
3.3. Modelo de Programación CUDA C . . . . .	23
3.3.1. Kernels . . . . .	23
3.3.2. Jerarquía de Hilos . . . . .	23
3.3.3. Jerarquía de Memoria . . . . .	26
3.3.4. Programación heterogénea . . . . .	28

<b>4. SIFT en GPU</b>	<b>30</b>
4.1. Análisis de SIFT para su Paralelización en GPU . . . . .	32
4.2. Implementación . . . . .	34
4.2.1. Construcción del espacio escala y aproximación a un Laplaciano de Gaussiana . . . . .	35
4.2.2. Detección de puntos característicos . . . . .	37
4.2.3. Eliminación de puntos característicos malos . . . . .	38
4.2.4. Asignación de orientación a los puntos característicos . . . . .	39
<b>5. Pruebas y Resultados</b>	<b>42</b>
5.1. Pruebas . . . . .	42
5.2. Resultados . . . . .	44
<b>6. Conclusiones y Trabajo a Futuro</b>	<b>50</b>
6.1. Conclusiones . . . . .	50
6.2. Trabajo a Futuro . . . . .	51
<b>A. Kernel Convolución</b>	<b>52</b>
<b>B. Kernel Localización de máximos y mínimos</b>	<b>54</b>
<b>C. Kernel Remover puntos malos</b>	<b>56</b>
<b>D. Kernel Asignar magnitud y orientación</b>	<b>58</b>
<b>E. Kernel Puntos Caractristicos</b>	<b>60</b>
<b>Bibliografía</b>	<b>62</b>
<b>Índice de figuras</b>	<b>64</b>

# CAPÍTULO 1

## INTRODUCCIÓN

### 1.1. Contexto

El software de los robots móviles en ocasiones es más complejo que el hardware que lo conforma. La complejidad del software dependerá de cuánta interacción tenga el robot con su entorno. Una parte esencial del software es la forma de interpretar y darle significado a los datos obtenidos de los diferentes sensores del robot.

La vista es a lo que más recurre un ser humano para obtener información de su entorno. Por ello no es de extrañarse que la visión computacional tenga una participación muy importante en la robótica, porque estas máquinas empiezan a ser utilizadas para tareas que antes solo los humanos realizaban, mismas que deben realizar de una manera adecuada y en tiempo.

El tiempo en el que se realizan las actividades en cualquier ámbito siempre ha sido importante. El paralelismo es la forma por la cual optaron los nuevos procesadores para mejorar su velocidad, ya que solo incrementar el número de pulsos de reloj por segundo implica problemas como la disipación de calor. Pero no todo es procesadores multinúcleo, las unidades de procesamiento gráfico por sus siglas en inglés GPU se pueden programar para realizar tareas de propósito general.

## 1.2. Problema a resolver

Los algoritmos que se utilizan en visión computacional aunque confiables consumen mucho tiempo de procesador, por lo cual se ha tratado de hacer más eficientes estos algoritmos, pero provocando que la confiabilidad disminuya. El tiempo en el cual se adquiere y procesa la información es crucial en la actividad de un robot, pues de esto depende que decisión tomara.

Con base en lo anterior, tan importante es el tiempo en que procesemos los datos, para tomar una decisión, como importante es que la información obtenida sea congruente. En muchos casos, el software que se ejecuta en paralelo es más rápido que el secuencial. Los algoritmos que se manejan en visión computacional son casi siempre secuenciales. Otro punto importante es que recursos como el procesador y la memoria de la computadora del robot siempre estarán siendo demandados por otros módulos del robot.

## 1.3. Hipótesis

La finalidad del presente documento es confirmar la siguiente hipótesis:

*"Por medio del uso de unidades de procesamiento gráfico de propósito general tener un mejor desempeño en la ejecución del algoritmo SIFT paralelo, comparado con una implementación secuencial del mismo"*

Respecto al alcance, se considerara valida la hipótesis si se pueden obtener los puntos característicos de una imagen con los cuales se podrían encontrar descriptores para su comparación. El desempeño se mide comparando el tiempo de ejecución entre el sistema actual del robot y el propuesto.

## 1.4. Estructura de la tesis

En el capítulo Marco Teórico se consideran dos puntos importantes que son la extracción y descripción de las características de una imagen. Aunado a esto se explica el proceso para extraer puntos característicos y obtener el descriptor de cada uno de estos de una manera que sean tolerantes a diferentes transformaciones por medio del algoritmo de SIFT, y como es que han venido cambiado los procesadores hasta poder llegar a el computo en las GPU. El capítulo Unidad de Procesamiento de Gráficos de Propósito General se enfoca en cómo han cambiado los procesadores multinúcleo, la forma en la que los programamos y sobre todo el cómputo en cooperación con las tarjetas gráficas. Se enfoca en las GPU de la familia de nVidia, se comenta un poco de la historia de estos multiprocesadores, su arquitectura y el modo en que podemos programar estos dispositivos, que ya no son solo utilizados en gráficos.

En el Capítulo SIFT en GPU se presentan puntos importantes al momento de paralelizar un algoritmo: Cómo es que se propone dividir el algoritmo de SIFT para que sea más sencillo el proceso de paralelización, y como trabajan los kernels en general para este caso. Más adelante se explica cómo es que están estructuradas una a una las secciones, en las que dividimos el algoritmo.

Para el capítulo Pruebas y Resultados se compara el tiempo que toma obtener los puntos característicos SIFT con el sistema propuesto y algunos otros ya implementados como la librería OpenSIFT y OpenCV.

El capítulo Conclusiones y Trabajo a Futuro se tomaran en cuenta cuales fueron las ventajas y desventajas de haber paralelizado el algoritmo SIFT respecto a los resultados. Como es que benefician estos resultados y cuáles serían los puntos con los que se podría seguir adelante con este trabajo de investigación.

## CAPÍTULO 2

---

### MARCO TEÓRICO

#### 2.1. Extracción y descripción de características

Las características de los objetos son cualidades que sirven para identificarlos dentro de una imagen. Para poder distinguir objetos que estén en una imagen nos basamos en las características encontradas por algún algoritmo, mismas que serán encapsuladas en un descriptor. Un descriptor de un objeto es la representación, de una manera reducida, de todas las características que se pueden obtener del objeto. Esto facilitará la identificación de los diferentes objetos que existan en una imagen.

Para extraer las características existen diferentes métodos. Dependerá de que algoritmo se utilice pues cada uno de ellos se enfoca en encontrar diferentes características como esquinas, bordes, crestas y regiones que son más obscuras o más claras. No todos los reconocedores encontrarán las mismas características o todos los tipos de características. Para que estas características sean robustas deben poder ser encontradas aunque los objetos se encuentren rotados, escalados, con cambios de iluminación o si están parcialmente ocluidos.

A continuación se explicara cómo se obtienen estas características, a las cuales les llamaremos puntos característicos, por medio del algoritmo de SIFT.

### 2.1.1. Características invariantes a transformaciones afines

El algoritmo Scale-Invariant Feature Transform (SIFT), propuesto por Lowe en [1], provee un método robusto para la extracción de puntos característicos que se utilizan para generar el descriptor. Los puntos que se encuentran son invariantes a diferentes transformaciones como traslación, escalamiento y rotación. Han mostrado tener un amplio rango de tolerancia a transformaciones afines, adición de ruido y cambios de iluminación. A continuación se describirán los pasos del algoritmo para la generación del conjunto de puntos característicos:

#### Detección de puntos extremos en el Espacio-Escala

Se realiza una búsqueda en las imágenes en todo el espacio escala. Para localizar puntos extremos se debe identificar su ubicación y escala para volver a encontrarlos no importando la vista o tamaño del mismo objeto.

El espacio escala es un conjunto de imágenes, que se forman a partir de suavizar la imagen original a diferentes niveles de detalles, los cuales son definidos por un parámetro  $\sigma$ . Está representado por la función  $L(x, y; \sigma)$  la cual se forma por la convolución de  $G(x, y; \sigma)$  con la imagen original  $I(x, y)$ :

$$L(x, y; \sigma) = G(x, y; \sigma) * I(x, y)$$

Donde  $*$  es el operador convolución en  $(x, y)$ , y

$$G(x, y; \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

Para la detección de puntos extremos estables se aplica el espacio escala usando diferencias de gaussiana convolucionadas con una imagen en lugar de solo un filtro gaussiano  $D(x, y; \sigma)$  que podremos calcular por la diferencia de dos escalas cercanas separadas por un factor  $k$  multiplicativo:

$$\begin{aligned} D(x, y; \sigma) &= (G(x, y; k\sigma) - G(x, y; \sigma)) * I(x, y) \\ &= L(x, y; k\sigma) - L(x, y; \sigma) \end{aligned}$$

La diferencia de gaussiana es una aproximación muy cercana a el laplaciano de gaussiana (LoG) normalizado en escala,  $\sigma^2\nabla^2G$ . La normalización hecha con el factor  $\sigma^2$  es necesaria

## 2.1. EXTRACCIÓN Y DESCRIPCIÓN DE CARACTERÍSTICAS

---

para poder asegurar que el algoritmo será invariante a los cambios en tamaño. La relación entre  $D$  y  $\sigma^2 \nabla^2 G$  es una ecuación en derivadas parciales:

$$\frac{\partial G}{\partial \sigma} = \sigma \nabla^2 G$$

Podemos ver que  $\nabla^2 G$  se puede calcular con una aproximación de diferencias finitas de  $\frac{\partial G}{\partial \sigma}$  usando diferencias de escalas próximas de  $k\sigma$  y  $\sigma$ :

$$\sigma \nabla^2 G = \frac{\partial G}{\partial \sigma} \approx \frac{G(x, y, k\sigma) - G(x, y, \sigma)}{k\sigma - \sigma}$$

Y, por lo tanto:

$$G(x, y, k\sigma) - G(x, y, \sigma) \approx (k - 1)\sigma^2 \nabla^2 G$$

En la figura 2-1 se puede ver la construcción de  $D(x, y, \sigma)$ . La imagen inicial se convoluciona con diferentes máscaras gaussiana para producir imágenes separadas por un factor constante  $k$  en el espacio escala. Se divide cada octava del espacio escala entre un numero entero  $s$  de intervalos, quedando entonces  $k = 2^{\frac{1}{s}}$ . Se producen  $s + 3$  imágenes emborronadas en la pila por octava.

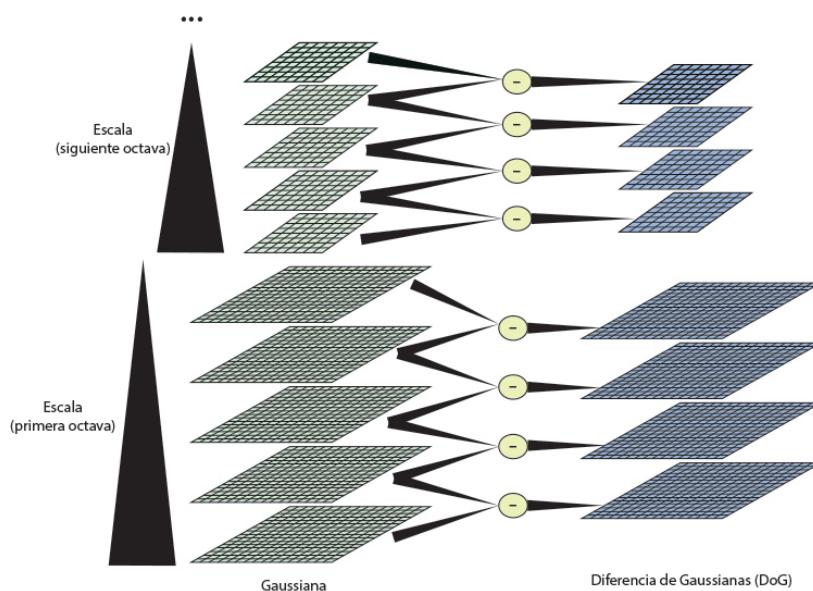


Figura 2-1: Espacio Escala de Diferencia de Gaussiana

## 2.1. EXTRACCIÓN Y DESCRIPCIÓN DE CARACTERÍSTICAS

---

Para extraer las ubicaciones máximas y mínimas (puntos extremos) en  $D(x, y, \sigma)$  cada punto es comparado con sus ocho vecinos en la misma imagen y con sus otros dieciocho vecinos de escala, nueve en la imagen de arriba y nueve en la imagen de abajo (Figura 2-2). Solo se selecciona el punto si es el más grande o el más pequeño de entre todos sus vecinos.

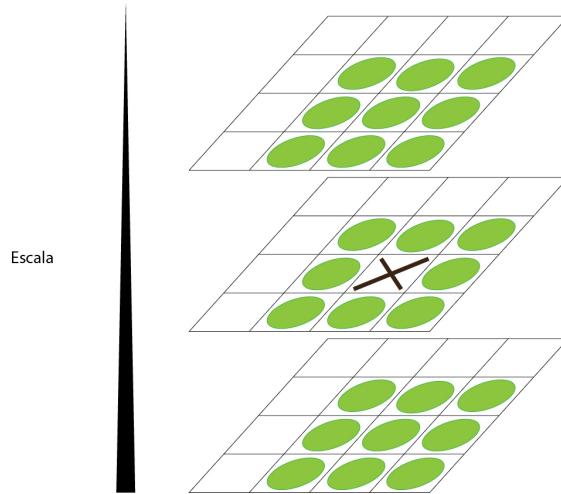


Figura 2-2: Espacio Escala de Diferencia de Gaussiana

### Localización de puntos característicos

Una vez que se seleccionaron los puntos extremos se aplica una medida de estabilidad sobre todos para descartar aquellos que no sean adecuados, para obtener puntos característicos de forma precisa. Existen dos casos donde los puntos extremos anteriormente seleccionados tienen que ser eliminados:

1. El punto tiene un contraste muy bajo.
2. El punto está localizado sobre un borde.

Para eliminar los puntos del caso uno primero se debe obtener la serie de Taylor del espacio escala  $D(x, y, \sigma)$ :

$$D(X) = D + \frac{\partial D}{\partial X} X + \frac{1}{2} X^T \frac{\partial^2 D}{\partial X^2} X$$

## 2.1. EXTRACCIÓN Y DESCRIPCIÓN DE CARACTERÍSTICAS

---

Donde la  $D$  y su derivada son evaluadas en el punto  $X = (x, y, \sigma)^T$ . Cuando se deriva esta función respecto a  $X$  y se iguala a cero podemos encontrar los valores extremos:

$$\hat{X} = -\frac{\partial^2 D}{\partial X^2}^{-1} \frac{\partial D}{\partial X}$$

La función que evaluará al punto extremo será,  $D(\hat{X})$ , la cual rechazará al punto si es de muy bajo contraste, la cual se obtiene de sustituir  $\hat{X}$  en  $D(X)$ :

$$D(\hat{X}) = D + \frac{1}{2} \frac{\partial D^T}{\partial X} \hat{X}$$

En el trabajo de Lowe [1], se encontró experimentalmente que cualquier valor extremo menor de 0.03 puede ser descartado:

$$|D(\hat{X})| < 0.03$$

Para el segundo caso, se utiliza una matriz Hessiana  $H$  de  $2 \times 2$  la cual se calcula en la escala y lugar del punto extremo:

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}$$

Los valores propios de  $H$  son proporcionales a las curvaturas de  $D$ . Si se toma prestado el criterio que se usa para la detección de esquinas usando el algoritmo de Harris [2], se puede evitar el cálculo de los valores propios ya que solo nos interesa su relación. Sea  $\alpha$  el valor propio de mayor magnitud y  $\beta$  el de menor. Entonces podemos calcular la suma de los valores propios de la diagonal de  $H$  y su producto por medio del determinante:

$$Tr(H) = D_{xx} + D_{yy} = \alpha + \beta,$$

$$Det(H) = D_{xx}D_{yy} - (D_{xy})^2 = \alpha\beta$$

Sea  $r$  la razón de la magnitud que existe entre  $\alpha$  y  $\beta$ ,  $\alpha = r\beta$ . Entonces:

$$\frac{Tr(H)^2}{Det(H)} = \frac{(\alpha + \beta)^2}{\alpha\beta} = \frac{(r\beta + \beta)^2}{r\beta^2} = \frac{(r+1)^2}{r}$$

## 2.1. EXTRACCIÓN Y DESCRIPCIÓN DE CARACTERÍSTICAS

---

El cual solo depende de la razón de los valores propios y no de los valores individuales. El valor de  $\frac{(r+1)^2}{r}$ , es más pequeño cuando los valores propios son iguales e incrementa con  $r$ .

Entonces para corroborar que la razón de las curvas principales es menor que cierto umbral  $r$  sólo se necesita:

$$\frac{Tr(H)^2}{Det(H)} < \frac{(r+1)^2}{r}$$

En la publicación de Lowe [1] se encontró un valor experimental para  $r = 10$  que elimina los puntos extremos que tengan la razón entre las dos curvas mayor que 10.

### Asignación de orientación

Por medio de la asignación de una orientación a cada punto característico, basado en propiedades locales de la imagen, el descriptor que encontraremos será invariante a la rotación. La ubicación en el espacio escala del punto característico, es usada para seleccionar la imagen suavizada por una máscara gaussiana,  $L$ , esto provocará que sea invariante a la escala. Para cada muestra de la imagen  $L(x, y)$  la magnitud del gradiente  $m(x, y)$  y la orientación  $\theta(x, y)$  son precalculadas por medio de diferencias de gaussiana:

$$m(x, y) = \sqrt{(L(x+1, y) - L(x-1, y))^2 + (L(x, y+1) - L(x, y-1))^2}$$

$$\theta(x, y) = \tan^{-1} \left( \frac{L(x, y+1) - L(x, y-1)}{L(x+1, y) - L(x-1, y)} \right)$$

Se formará un histograma de orientaciones que tendrá la orientación de los gradientes calculados en una región alrededor del punto característico. El tamaño de esta muestra dependerá de la ubicación en el espacio escala en la que se encuentre el punto característico. El histograma de orientaciones tendrá 36 divisiones cubriendo los 360 grados.

Cada muestra agregada se pondera por la magnitud de su gradiente y por una máscara circular gaussiana ponderada con  $1.5 \times \sigma$ , donde la  $\sigma$  dependerá del espacio escala donde reside el punto característico.

Los picos en el histograma de orientación corresponden a las direcciones dominantes de los gradientes locales. Se encuentra el pico más grande y cualquier otro pico que se encuentre en el rango de 100 % – 80 % del pico más grande se utiliza para hacer que el punto característico

## 2.1. EXTRACCIÓN Y DESCRIPCIÓN DE CARACTERÍSTICAS

---

tenga una orientación (figura 2-3). Para ubicaciones con varios picos de magnitudes similares se generaran puntos característicos con la misma ubicación y escala pero con diferentes orientaciones. Solo el 15 % de los puntos se les asignan múltiples orientaciones, pero aun así esto contribuye mucho al momento de emparejar. Finalmente se obtiene una parábola usando como puntos tres picos cercanos entre sí, para interpolar la posición del pico con más precisión.

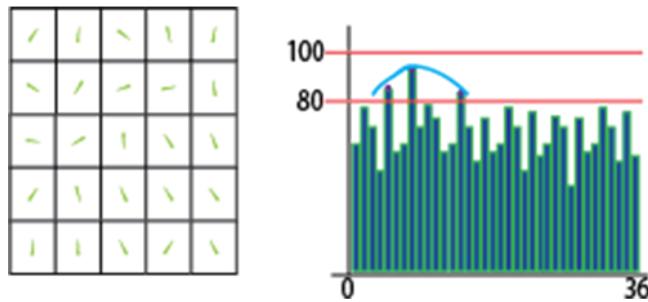


Figura 2-3: Histograma de Orientación

### Descriptor de puntos característicos

Hasta este momento se tiene una colección de puntos característicos, los cuales están formados por una ubicación, una escala y una orientación. Ahora debemos formar un descriptor que sea lo suficientemente distintivo. Para esto tenemos que tomar una muestra de la imagen al rededor del punto característico de  $16 \times 16$  pixeles que se dividirá en una región de  $4 \times 4$ . Se generará un histograma de orientación de los gradientes de cada región a diferencia del histograma de orientación explicado anteriormente, el histograma sólo tiene 8 divisiones con las cuales se cubrirán los 360 grados. Igualmente se usará una ponderación gaussiana para la asignación de la magnitud al histograma (figura 2-4).

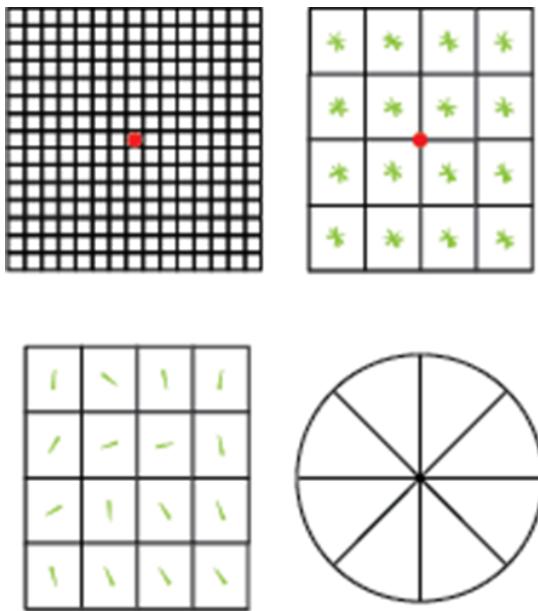


Figura 2-4: Descriptor

Al final, el descriptor de cada punto característico estará formado por un vector que tiene las ocho orientaciones de los  $4 \times 4$  histogramas. Por lo tanto el tamaño del vector será de  $4 \times 4 \times 8 = 128$  elementos.

## 2.2. Cómputo en GPU

Hasta hace 12 años la velocidad a la que crecía cada generación de procesadores mononúcleo era increíble, los programas eran tan rápidos como cada nueva generación de procesadores. Este crecimiento entre cada generación se detuvo. El problema es que el consumo de energía y la disipación de calor no permiten aumentar la frecuencia de reloj del procesador y el nivel de actividades por ciclo, en una sola unidad de procesamiento (CPU). Todos los productores de procesadores migraron a un nuevo modelo, los procesadores multinúcleo incrementaron el poder de procesamiento.

Este cambio en los procesadores tuvo un gran impacto a los programadores, la mayoría de las aplicaciones son escritas de forma secuencial porque la ejecución será paso a paso como esta descrito en el código, esto permitirá que al momento de corregir errores de programación o entender cómo funciona el programa sea más sencillo.

Pero un programa secuencial ejecutándose en un solo núcleo del procesador no será más rápido. Entonces los programadores ya no pueden agregar cualidades y capacidades a sus programas de forma tradicional.

Llega el momento de cambiar. Si se desea que la calidad de los programas siga escalando con cada generación de procesadores se deben crear programas que trabajen con múltiples hilos, cooperando todos para completar un trabajo más rápido.

Existen dos corrientes principales en cuanto a los procesadores multinúcleo: el primero es donde se pretende mantener la velocidad de los programas secuenciales mientras se mueven entre múltiples núcleos. La segunda se centra más en la ejecución de aplicaciones que fueron específicamente diseñadas para aprovechar todos los núcleos en el procesador al mismo tiempo, los procesadores de esta corriente tienen un gran número de núcleos pequeños que va creciendo con cada generación. Es esta rama en la que entran las unidades de procesamiento gráfico o por sus siglas en inglés GPU [3]. En la figura 2-5 podemos ver una comparación de estas dos corrientes considerando a la cantidad de operaciones de punto flotante que pueden realizar en un segundo (flops).

En el capítulo 3 hablaremos más a detalle sobre el cómputo en GPU especialmente de los GP-GPU de nVidia.

## 2.2. CÓMPUTO EN GPU

---

### Theoretical GFLOP/s

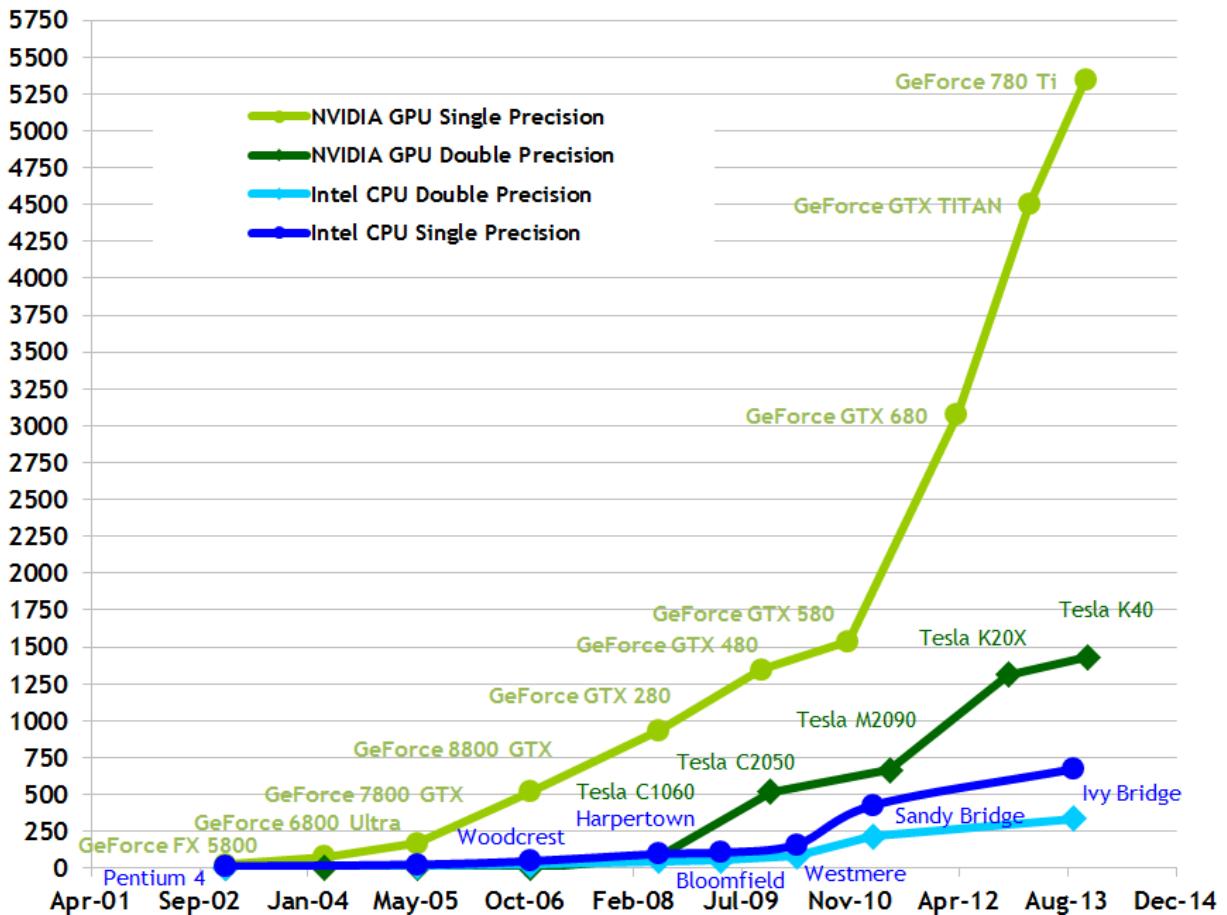


Figura 2-5: Operaciones en Punto Flotante por segundo de GPU y CPU [4]

## CAPÍTULO 3

### UNIDAD DE PROCESAMIENTO DE GRAFICOS DE PROPOSITO GENERAL

”Las GPU han evolucionado al punto que muchas aplicaciones del mundo real se están implementando fácilmente en ellas y se ejecutan muchísimo más rápido que en sistemas con múltiples núcleos. Las arquitecturas de computación del futuro serán sistemas híbridos con GPU de núcleos paralelos trabajando en tandem con CPU de múltiples núcleos” (figura 3-1).[5]



Figura 3-1: Sistema Híbrido con GPU y CPU

### 3.1. Breve Historia de las GPU

La necesidad de mejorar los gráficos para los videojuegos provocó un gran avance en el diseño de hardware. En la década de los ochentas las tarjetas gráficas dedicadas, no eran más que pipelines fijos que desplegaban formas geométricas calculadas por el CPU por medio del hardware de acceso directo a memoria (DMA). Esto les daba un funcionamiento fijo y solo eran configurables mediante dos librerías: OpenGL de *Silicon Graphics* y Direct3D de *Microsoft*.

A finales de los noventas el hardware se volvió más programable, un ejemplo de estas tarjetas gráficas es la GeForce 256[6] fue lanzada al mercado en 1999, a la que se le acuño el nombre de GPU. La cual aportó características graficas importantes, como por ejemplo: funciones de transformación, iluminación, organización y rendering, la capacidad de procesar 15 millones de triángulos por segundo y un rendimiento de 480 millones de píxeles por segundo. Además su motor de rendering 256 bits mostró una mejora en cuanto a la complejidad visual.

Esta tecnología revolucionaria llamó la atención de otros profesionales, además de artistas y desarrolladores de video juegos, quienes utilizaron el gran rendimiento de punto flotante que tenían los GPU para otros objetivos por ejemplo procesar imágenes médicas, exploración sísmica, centros de supercómputo, entre otras[7]. De esta forma surge un movimiento para utilizar para fines diferentes a los gráficos a las GPU.

En ese momento, la GPU de propósito general (GP-GPU) era muy difícil de manipular: solo aquellos que tenían amplios conocimientos en lenguajes de programación de gráficos desarrollaban para estas plataformas. Puesto que los cálculos para resolver los problemas generales debían ser representados por triángulos o polígonos.

Fue hasta 2001, en la Universidad de Stanford, cuando un equipo liderado por Ian Buck se propuso ver el GPU como un *procesador de flujos*. Este equipo desarrollaría *Brook* [8], un lenguaje de programación diseñado para tener la misma sintaxis de C, con algunas características adicionales. Y con el objetivo de minimizar el complejo trabajo de análisis que se requería para generar aplicaciones paralelas. En este, se introdujeron conceptos como los flujos (streams), kernels y los operadores de reducción. Esto generó un gran impulso a los GPU como procesadores de propósitos generales, principalmente por dos razones: pues se trataba

### 3.2. PLATAFORMA Y MODELO DE PROGRAMACIÓN DE CÓMPUTO PARALELO

de un lenguaje de más alto nivel. Lo más importante, los programas escritos en *Brook* eran hasta 7 veces más rápido que códigos similares existentes.

La compañía NVIDIA se dio cuenta que tenía un hardware muy poderoso en las manos, sin embargo debía complementarlo con herramientas de hardware y software intuitivas. Invitó a Ian Buck a colaborar con ellos. El objetivo sería ejecutar C a la perfección en una GPU. NVIDIA alcanzó este objetivo en 2006 con el lanzamiento de CUDA, la cual sería la primera solución para las GP-GPU. Aunado a esta solución, lanzó la GeForce 8800, la cual fue diseñada para ser usada en cómputo de propósito general con su arquitectura inspirada en la de CUDA.

## **3.2. Plataforma y modelo de programación de cómputo paralelo**

NVIDIA lanzó en noviembre de 2006 una Arquitectura Unificada de Dispositivos de Cómputo (CUDA), es una plataforma para cómputo paralelo y un modelo de programación, que permite obtener aumentos en los rendimientos del cómputo gracias a la ayuda que la unidad de procesamiento de gráficos le proporciona al CPU.

Los dispositivos CUDA aceleran la ejecución de los programas que procesan una gran cantidad de datos ya que la arquitectura de esta plataforma, es similar a un procesador tradicional de computadora pero, tienen la cualidad de que los procesadores son masivamente paralelos, equipados con una gran cantidad de unidades aritméticas. En estas unidades aritméticas se ejecuta la misma instrucción y de acuerdo con la taxonomía de Flynn, pertenecen a la categoría de *una instrucción, múltiples datos* (SIMD) como se puede ver en la figura 3-2.

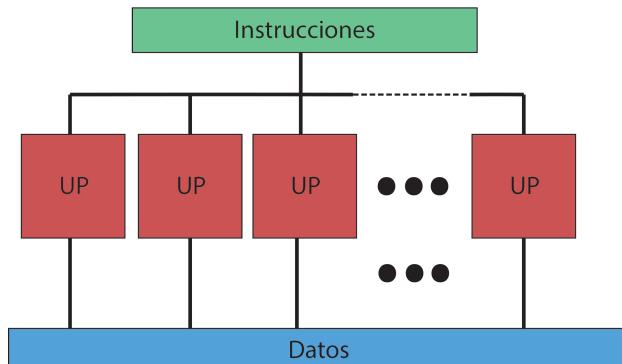


Figura 3-2: SIMD

El modelo de programación para desarrollar programas para las GPU, es una extensión del lenguaje C conocida como CUDA C. Actualmente existen alternativas a CUDA C tales como FORTRAN, Python, .NET (combinando CUDA con Microsoft's F#), o alguna API como OpenCL u OpenACC[9].

#### **3.2.1. Arquitecturas**

Como se describió anteriormente la arquitectura de CUDA fue diseñada para que la GPU pudiera ser utilizada en aplicaciones de propósito general. Ésta tiene un arreglo de procesadores con múltiples unidades aritmético-lógicas (ALU), las cuales fueron diseñadas para poder realizar operaciones de punto flotante, cumpliendo los requisitos del Instituto de Ingeniería Eléctrica y Electrónica (IEEE). Además de esto, las ALU deben tener acceso a diferentes tipos de memoria, como la compartida entre unidades y la memoria de la tarjeta gráfica. Estas ALU tan particulares en la arquitectura de CUDA se conocen como *CUDA cores* y conforman gran parte de los Streaming Multiprocessor (SM). Los SM son procesadores que tienen la tarea de ejecutar los hilos concurrentemente. Además, los CUDA cores están formados por una memoria cache (shared memory), registros y unidades de funciones especiales.

#### **Arquitectura Fermi**

Los GPU basados en la arquitectura Fermi [10], están formados por 512 CUDA cores. Los CUDA cores ejecutan operaciones de punto flotantes o enteras por ciclo de reloj en cada hilo. Los 512 CUDA cores están organizados en 16 SM de 32 cores cada uno (figura 3-3). El GPU

### 3.2. PLATAFORMA Y MODELO DE PROGRAMACIÓN DE CÓMPUTO PARALELO

tiene seis particiones de memoria de 64-bits, una capacidad de leer 384-bits de la memoria simultáneamente y una capacidad de hasta 6GB de memoria DRAM categoría DDR5. El sistema de conexión entre el GPU y el CPU es vía PCI-Express. La forma en que se hace la distribución del trabajo en cada bloque es decidida por el módulo llamado *GigaThread*, el cual pasa las tareas a cada SM para que haga la asignación de trabajo a cada hilo.

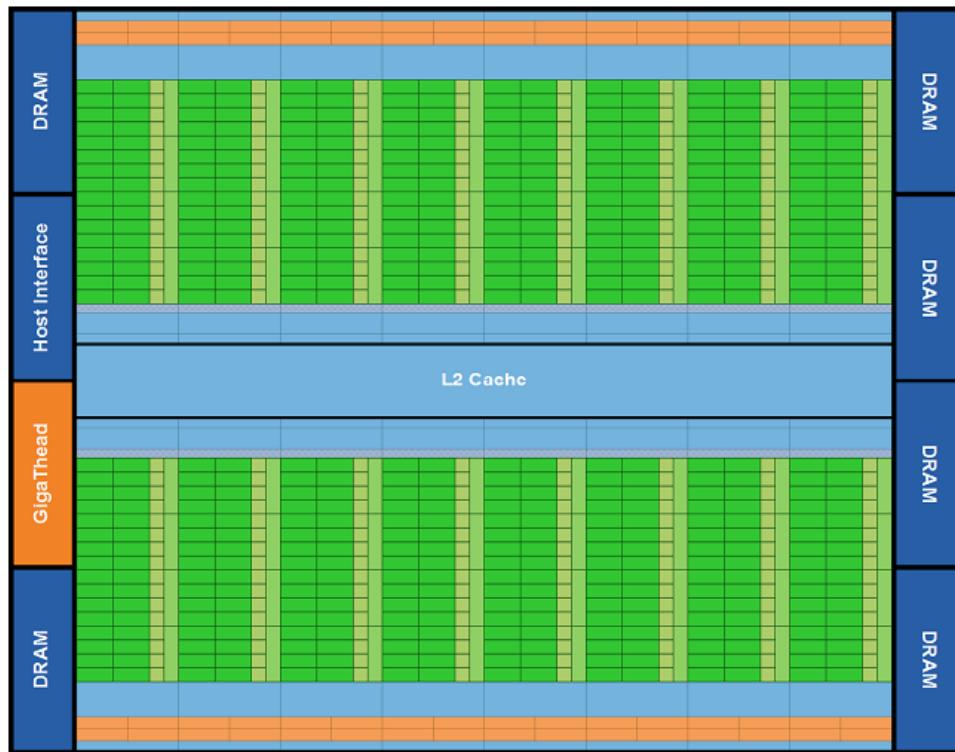


Figura 3-3: La arquitectura Fermi tiene sus 16 SM alrededor de la memoria compartida L2 cache [10] Ver figura 3-4 para detalle de los SM.

Esta arquitectura tiene cualidades significativas como el rendimiento en las operaciones de doble precisión dedicado a cómputo científico, soporte para corrección de errores para asegurar las operaciones con números muy grandes en aplicaciones delicadas. Se implementó una jerarquía en la memoria cache que permitió aumentar la eficiencia en cuanto a las lecturas a memoria, la memoria compartida tuvo un incremento y las operaciones atómicas incrementaron su desempeño gracias a que se aumentaron las unidades de operaciones atómicas y la aparición de la memoria cache L2.

Las SM de la arquitectura Fermi (figura 3-4) están formadas por diferentes elementos, inician-

### 3.2. PLATAFORMA Y MODELO DE PROGRAMACIÓN DE CÓMPUTO PARALELO

do por los 32 CUDA cores, cada uno con una unidad aritmética lógica para las operaciones con enteros y una unidad de punto flotante. Cumplen con la norma IEEE 754-2008 que permite realizar una multiplicación y una suma en un solo paso de redondeo. La asignación de trabajo en las SM se realiza por el modulo *GigaThread*, que divide en bloques de hilos a cada SM. Después los planeadores de *warps* dividen el trabajo de este bloque en grupos de 32 hilos para su ejecución dentro de la SM. Cuentan también con 16 unidades load/store, las cuales permiten calcular el origen y destino de los dieciséis hilos por pulso de reloj, y 4 unidades de funciones especiales (SFU), que ejecutan instrucciones complejas como cálculo de senos, cosenos, reciproco y raíz cuadrada.

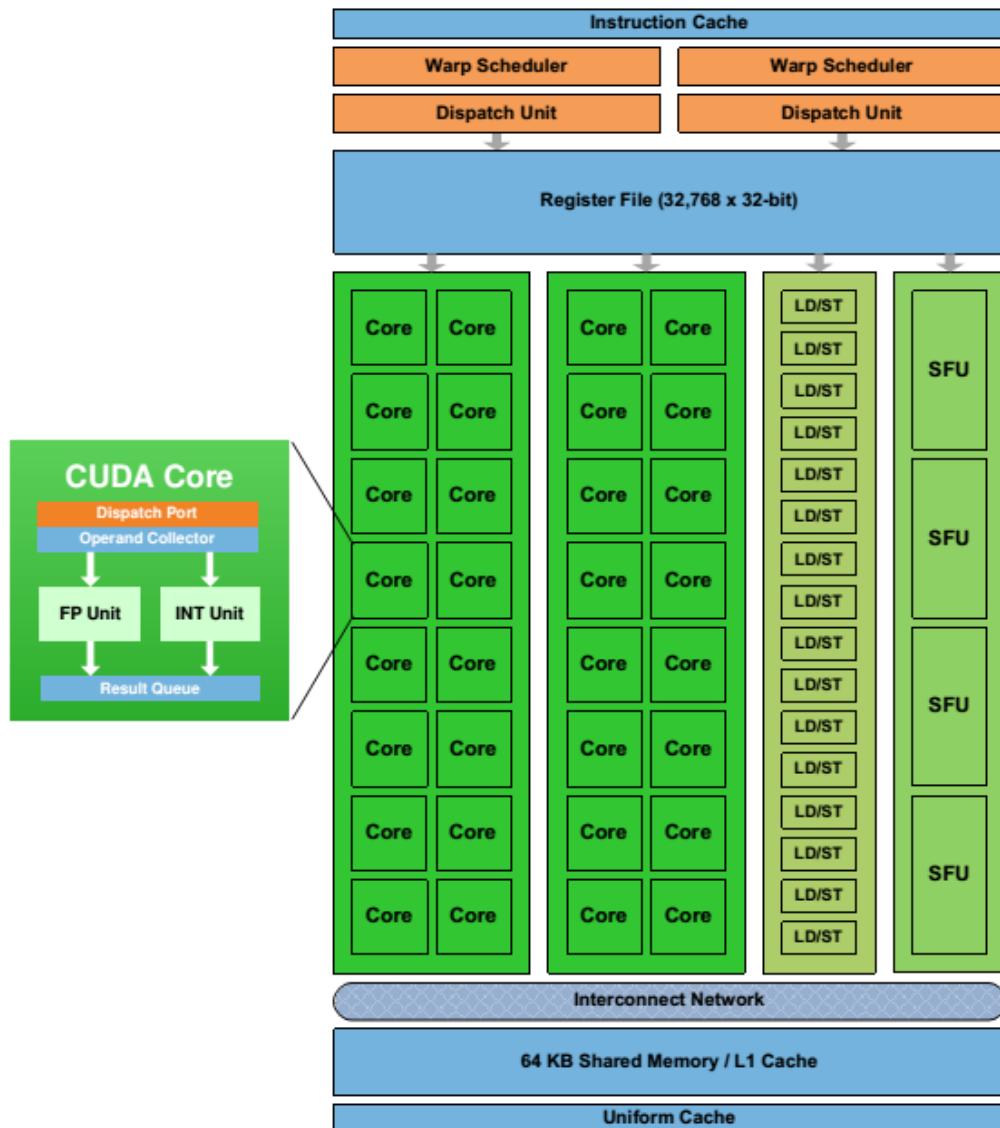


Figura 3-4: Fermi Streaming Multiprocessor (SM)[10]

### Arquitectura Kepler

La arquitectura Kepler [11], modificó los SM de su antecesor Fermi llamándolo Next Generation Streaming Multiprocessor (SMX) (figura 3-5). El nuevo procesador de esta arquitectura está formado por 15 de estos procesadores y 6 controladores de memoria de 64-bits. La cantidad de CUDA cores que contiene es de 192 de precisión simple, y 64 unidades de doble precisión.

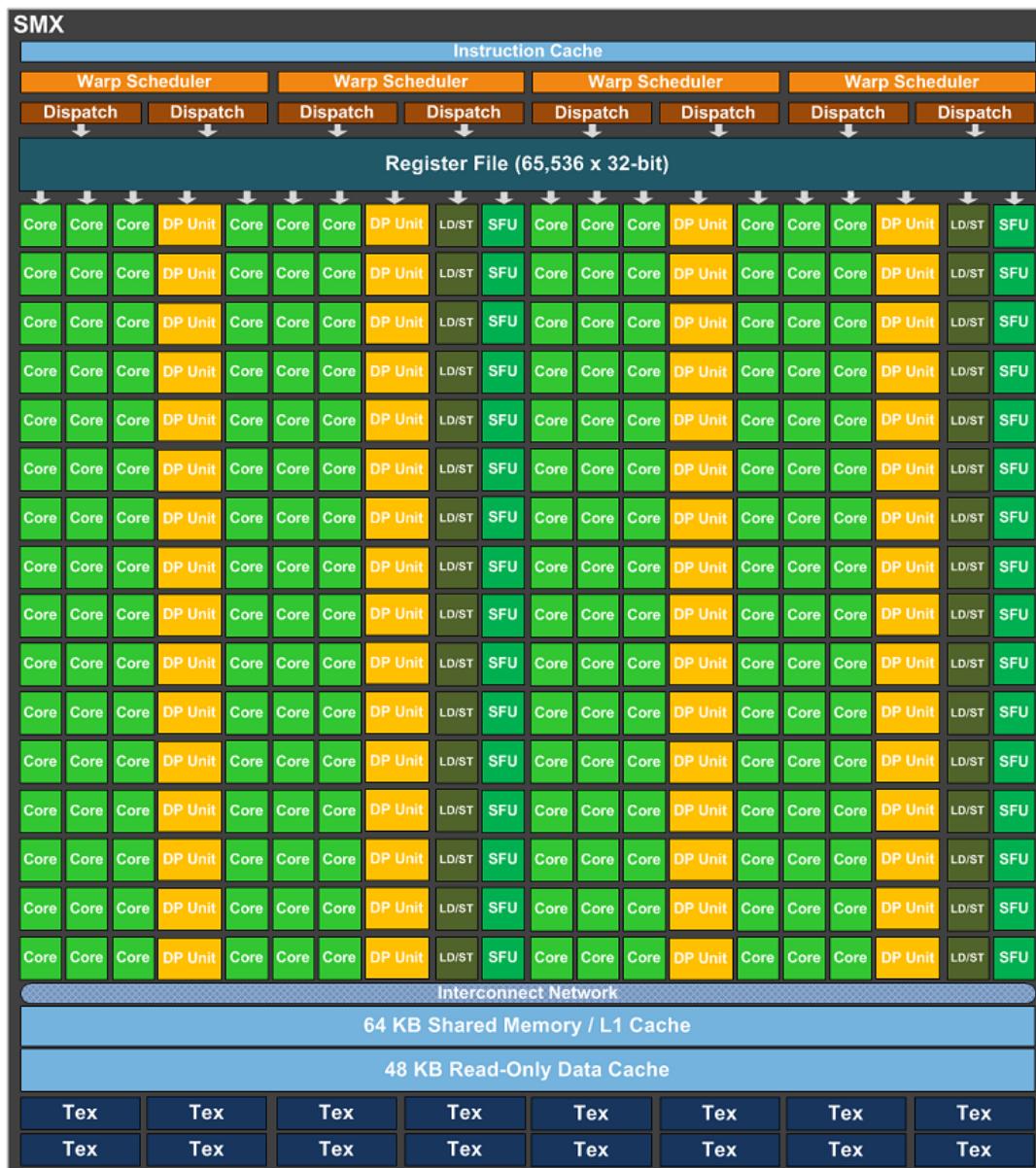


Figura 3-5: Kepler Next Generation Streaming Multiprocessor (SMX) [11]

### 3.2. PLATAFORMA Y MODELO DE PROGRAMACIÓN DE CÓMPUTO PARALELO

Las unidades load/store aumentaron a 32 y las SFU incrementaron a 32, ocho veces más que en la Fermi. La asignación de tareas a hilos dentro del SMX es un trabajo desempeñado por los planeadores de warps, los cuales cuenta con bloques de 32 hilos. Además, tiene 4 planificadores de warps, por lo que se tienen 2 unidades de despacho de instrucciones, las cuales permiten repartir y ejecutar los 4 warps de manera concurrente.

También cuenta con una memoria caché L1 con una capacidad de 64KB, configurable a 16, 32 o 48 KB para la memoria cache y el resto para la memoria compartida. Esto da 65536 registros por SMX, en los cuales, cada hilo puede tener acceso a 255 registros para el almacenamiento de datos. Integra una memoria de textura, la cual es un recurso valioso para programas donde se requiere probar o filtrar datos de una imagen. La memoria de textura en esta arquitectura dejó de ser un hardware dedicado solo a este objetivo y se creó un espacio en la memoria global de solo lectura de 48KB que funciona como una memoria caché para agilizar las lecturas.

En esta arquitectura se agregó una característica: no se requiere del CPU para lanzar programas en la GPU, es decir, la GPU tiene la capacidad de generar más carga de trabajo, administrar recursos y obtener resultados dentro de la misma GPU (en la zona de más interés), donde se pueda requerir más poder de cómputo.

### **Arquitectura Maxwell**

La arquitectura Maxwell[12], sufrió un cambio de diseño para aumentar drásticamente su desempeño. Entre sus cambios, se tienen los nuevos SM llamados SM Maxwell (SMM) (figura 3-6), se redujo el número de CUDA cores a 128 y se separaron en 4 divisiones de 32. Cada una de esas divisiones tiene un planificador de warps para su bloque, el cual es capaz de despachar dos instrucciones por ciclo de reloj. Estas divisiones permitieron que se utilizara de una manera más eficiente el espacio y la energía gastada para el manejo de la transferencia de datos.

La memoria compartida se incrementó a 96KB y ya no se compartió con la memoria caché L1. Esta última comparte espacio con la memoria de textura. Los registros, las SFU, y las unidades load/store siguieron siendo la misma cantidad.

### 3.2. PLATAFORMA Y MODELO DE PROGRAMACIÓN DE CÓMPUTO PARALELO

---

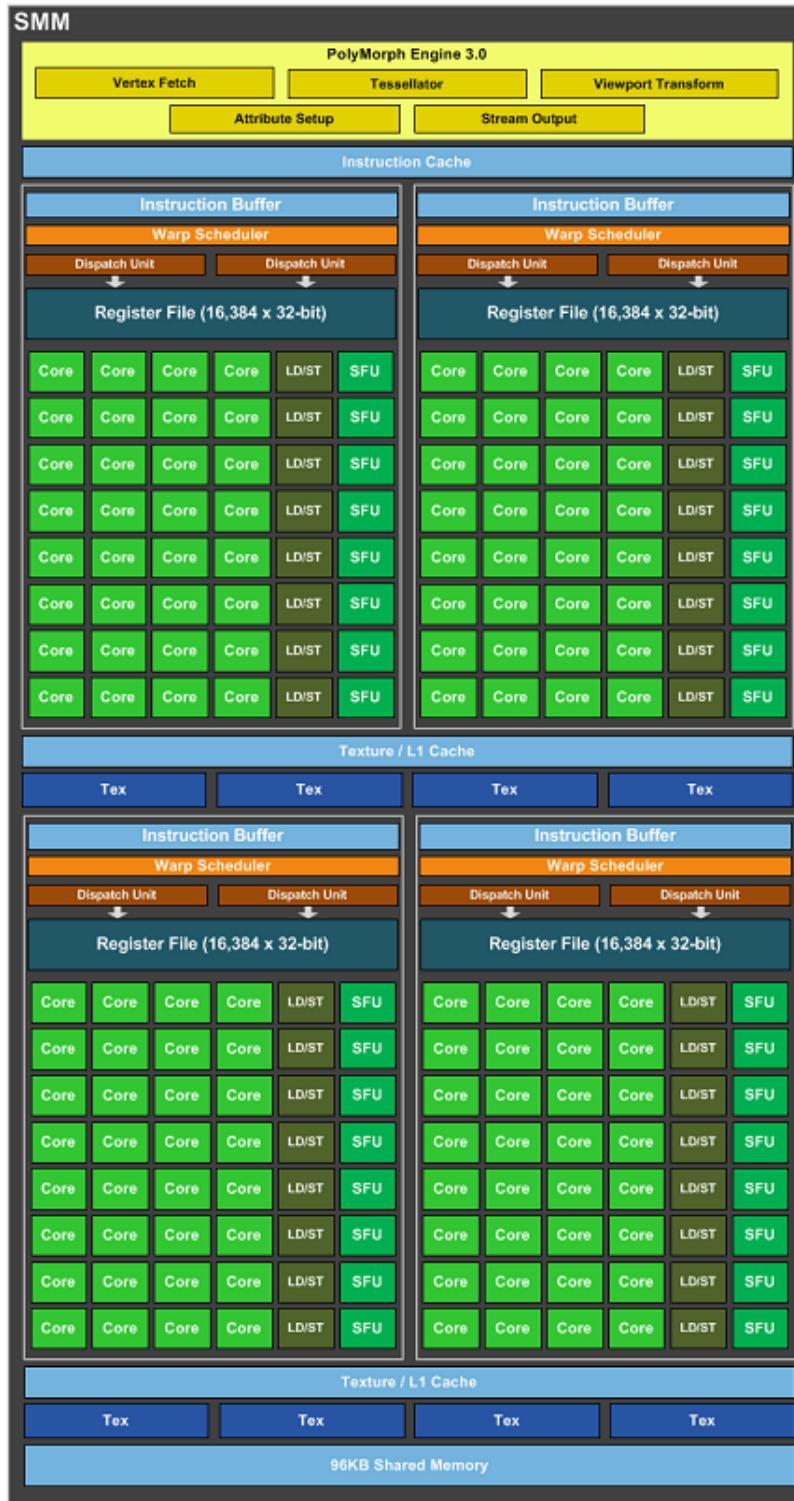


Figura 3-6: Maxwell Streaming Multiprocessor (SMM) [12]

### 3.3. Modelo de Programación CUDA C

La extensión del lenguaje C que proporciona CUDA para programar aplicaciones basadas en su arquitectura ofrece a los programadores familiarizados con este lenguaje una manera sencilla de escribir programas para ser ejecutados en la GPU. A continuación se explicará el núcleo del conjunto de instrucciones de esta extensión.

#### 3.3.1. Kernels

CUDA C, permite definir funciones llamadas *kernels* las cuales, cuando son llamadas, se ejecutan N veces en paralelo en N diferentes *hilos* CUDA. Para definir un kernel se usa la declaración `__global__`. Estos kernels se ejecutan en un dispositivo: la tarjeta gráfica instalada en la computadora, y se invocan por medio del equipo anfitrión. Este anfitrión no es más que el procesador que estará usando la tarjeta gráfica como coprocesador. El siguiente código muestra cómo se declara un kernel:

```
1 __global__ void kernel( ... )
2 {
3     ...
4 }
```

Código 3.1: Declaración de un Kernel en CUDA C.

Al lanzar un kernel desde el anfitrión, se debe escoger una configuración para los hilos CUDA que se lanzaran para ejecutarlo, dándole a cada uno de estos un identificador único (*threadID*).

#### 3.3.2. Jerarquía de Hilos

La configuración que se usa para lanzar los hilos, se especifica entre `<<< y >>>`. Se requiere de dos parámetros para el lanzamiento: el primero es la dimensión de la malla (véase figura 3-7), que se refiere al número de bloques, y el segundo es la dimensión del bloque, que es el número de hilos.

### 3.3. MODELO DE PROGRAMACIÓN CUDA C

---

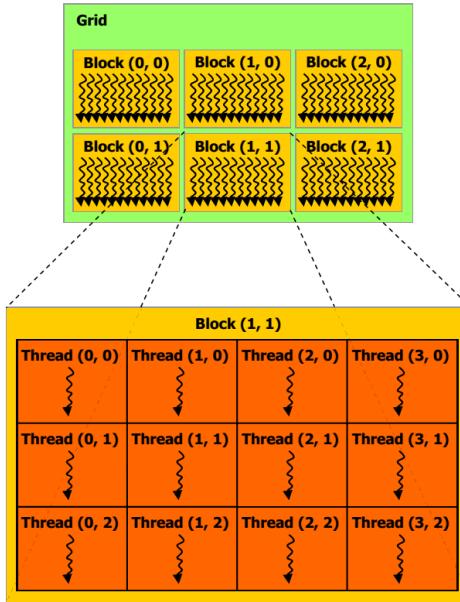


Figura 3-7: Organización bloques en malla e hilos en bloques [4]

Cada hilo tiene un identificador al que se puede acceder mediante el identificador **threadIdx**, el cual es un vector de tres componentes, por lo que, los hilos pueden usar identificadores de uno, dos o tres dimensiones para formar bloques unidimensionales, bidimensionales o tridimensionales. Los bloques están organizados en una malla que puede tener una, dos o tres dimensiones, y tienen un identificador al cual se puede acceder usando la variable **blockIdx**. Existe otra variable importante **blockDim** la cual especifica la dimensión del bloque.

A continuación se presenta un ejemplo de cómo se lanza un kernel:

```

1 __global__ void miKernel( ... )
2 {
3     ...
4 }
5 int main(....)
6 {
7     ...
8     dim3 gridDim(..., ..., ...);
9     dim3 blockDim(..., ..., ...);
10    miKernel<<<gridDim,blockDim>>>(...);
11    ...
12 }
```

Código 3.2: Lanzamiento de un Kernel en CUDA C.

### 3.3. MODELO DE PROGRAMACIÓN CUDA C

---

Una parte fundamental del paralelismo es la comunicación entre cada proceso/hilo que se ejecuta simultáneamente. Hacer cooperar los hilos en la GPU no es tarea fácil. Para la comunicación entre hilos se tiene la memoria compartida la cual permite el intercambio de datos solo entre hilos del mismo bloque. En cuanto a la sincronización de los hilos, se tiene una función llamada `__syncthreads()`, la cual sincroniza los hilos por barrera. Esto quiere decir que un hilo no puede seguir ejecutando su tarea hasta que todos los demás hilos lleguen a la misma instrucción. Esta característica solo aplica para hilos del mismo bloque. En la figura 3-8 se puede observar cómo se asignan los bloques a cada SM. Estos podrían asignarse en cualquier orden y ejecutarse en tiempos diferentes. De este modo, la sincronización de los hilos y la escritura/lectura a memoria compartida se permite, con la característica de que solo los hilos de un mismo bloque pueden ser sincronizados o comunicarse por medio de memoria compartida.

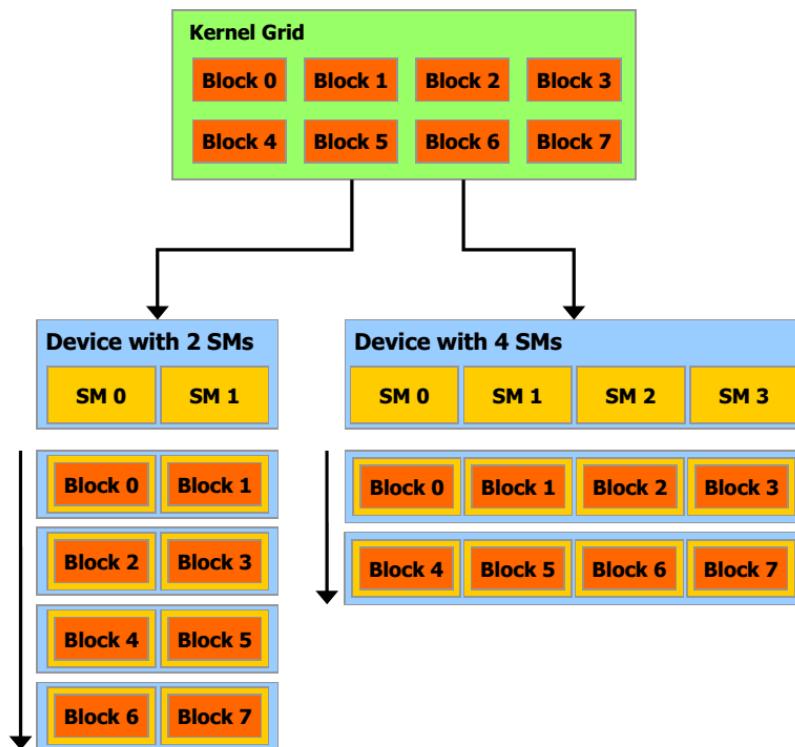


Figura 3-8: Asignación de bloques por SM [4]

### 3.3.3. Jerarquía de Memoria

La arquitectura CUDA cuenta con diferentes tipos de memoria, de los cuales se pueden leer los datos para operar y escribir los resultados obtenidos por los hilos, los cuales, son los encargados de realizar las operaciones sobre los datos. Cada uno de estos bancos de almacenamiento tiene características diferentes que, utilizados adecuadamente, pueden ayudar a mejorar el desempeño de los programas. A continuación se hablará de los tipos de memoria que se encuentran en las GPU (figura 3-9).

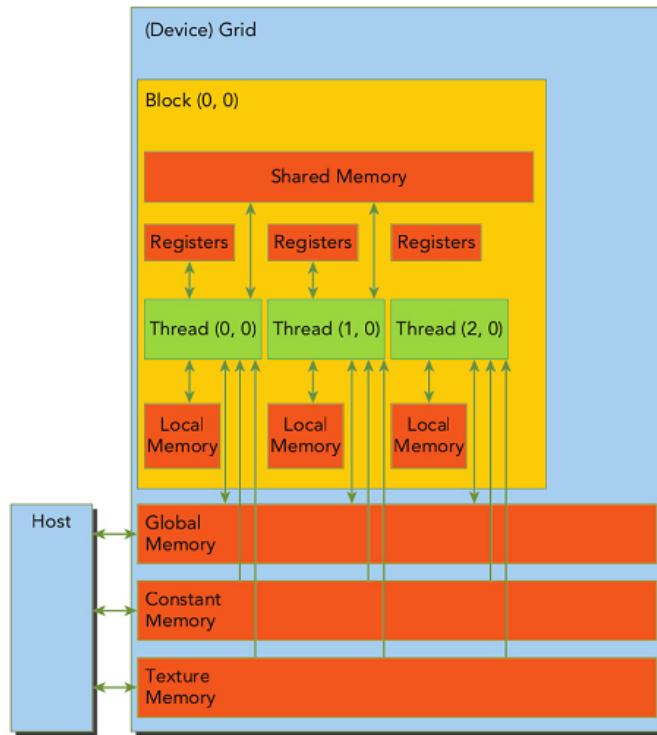


Figura 3-9: Tipos de Memoria de la GPU [13]

Los *registros* son el tipo de memoria con el tiempo de lectura/escritura más rápido en el dispositivo. En cada SM, hay miles de registros y los cuales son asignados en una cantidad fija a cada hilo cuando es lanzado un kernel. Estos son de 32-bits y pueden almacenar datos de tipo flotante o entero. La manipulación de los registros esta administrado por el sistema. La *memoria local* es un espacio de memoria privada que cada hilo tiene. En esta se almacenan los datos que no pudieron ser almacenados en los registros por ejemplo las variables

### *3.3. MODELO DE PROGRAMACIÓN CUDA C*

---

locales, las llamadas a funciones y el contexto de ejecución. Al igual que los registros, esta memoria es administrada por el sistema.

La *memoria compartida* es una memoria de tipo caché que se comparte entre hilos de un mismo bloque. Esta permite que los hilos de un bloque puedan comunicarse escribiendo y leyendo en ella para cooperar en la realización de un mismo objetivo. Su caché es especial ya que el programador define su manejo. La forma en la que se declara una variable en este espacio es mediante la palabra reservada `_shared_`. La latencia en esta memoria es hasta 100 veces menor que en la memoria global.

La *memoria constante* es una memoria de solo lectura, la cual alberga datos que no cambian a lo largo de la ejecución del kernel. Se puede usar esta memoria en el dispositivo igual que la memoria global, pero esta optimizada para enviar datos de lecturas a múltiples hilos. Esto se logra gracias a diferentes instrucciones que permiten el acceso a este cache de una forma más eficiente. Se puede declarar dicha variable con la palabra reservada `_constant_`, pero su contenido debe ser asignado por el anfitrión en la memoria del dispositivo, antes de lanzar el kernel mediante la función **cudaMemcpyToSymbol()**.

La *memoria de textura*, al igual que la memoria constante, es una memoria de solo lectura. Está diseñada para trabajar con estructuras llamadas CUDA array las cuales permiten lecturas eficientes en arreglos de hasta tres dimensiones. Las lecturas en este tipo de memoria tienen ventajas, como las diferentes formas de acceso o las interpolaciones de los datos, que se pueden utilizar sin costos adicionales de tiempo.

La *memoria global* es la memoria de lectura/escritura de mayor capacidad en la tarjeta gráfica, llegando al orden de los gigabytes. Las funciones que tiene son lectura de datos y escritura de resultados. También funciona como interfaz entre la GPU y el CPU. La persistencia de los datos en esta memoria es hasta que se liberan, lo que permite compartir datos entre kernels. Los hilos pueden acceder en cualquier momento a esta, pero su latencia es tan alta que puede provocar que sea más tardada la lectura de datos que los cálculos que se quieren realizar. Las funciones para reservar, manejar y liberar el espacio en memoria, desde el anfitrión, son: **cudaMalloc()**, **cudaMemCpy()** y **cudaFree()**.

### 3.3.4. Programación heterogénea

Para entender el modelo de programación de CUDA, es pertinente definir que código ejecutará el CPU y que código el GPU, a fin de que estos puedan trabajar en conjunto. El CPU es el equipo anfitrión (Host) el que decidirá cuándo es necesario usar al dispositivo GPU (Device), el anfitrión y el dispositivo también tendrán memorias separadas. Un Programa en CUDA C se ejecuta tal como se muestra en la figura 3-10.

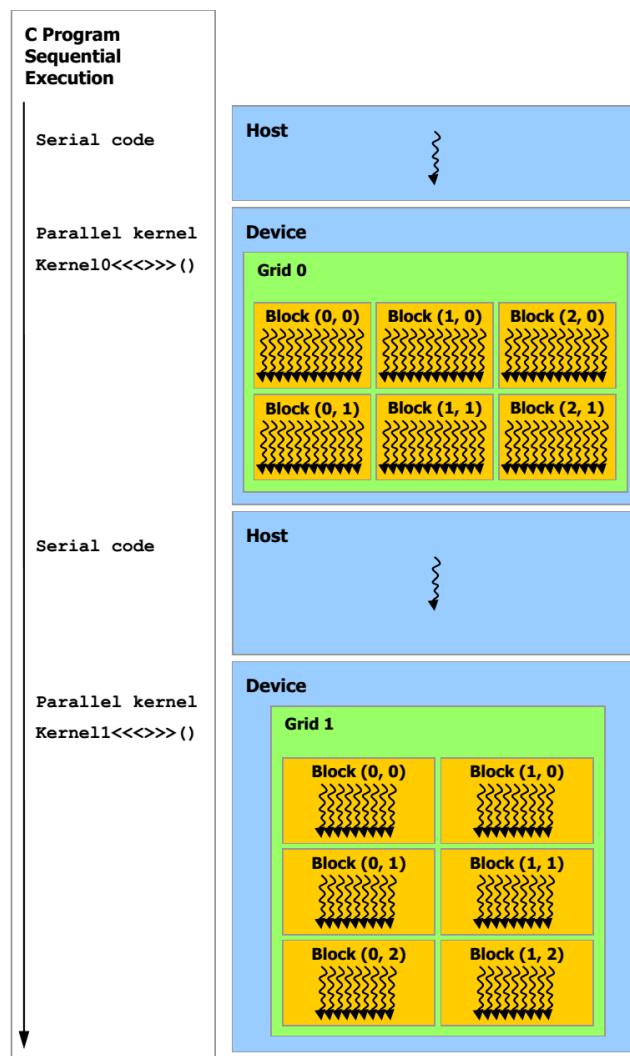


Figura 3-10: Programación Heterogénea [4]

En el código 3.3 de ejemplo se puede ver la estructura básica de un programa, y cómo se declaran y definen funciones kernel. También se puede ver que hay funciones ejecutables en

### 3.3. MODELO DE PROGRAMACIÓN CUDA C

---

el GPU que son llamadas desde algún kernel definidas con la palabra reservada `__device__`. En la función principal, el anfitrión se encarga de obtener los datos que se le proporcionaran al kernel y almacenarlos para ser procesados por el GPU. También se puede ver cómo es que se reserva la memoria en el dispositivo para los datos de entrada y salida que el kernel necesite para procesarlos.

Después de realizar una copia de los datos del anfitrión al dispositivo, se pueden ejecutar uno o más kernels en la GPU. Una vez finalizada la ejecución de sus kernels, el resultado se copia a la memoria del anfitrión. Al final, sólo queda liberar los recursos que ya no serán utilizados.

```
1 __device__ L funcionDevice()
2 { ...
3     __global__ void KernelUno(L*, ... )
4 {
5         ...
6         L r= fooDevice();
7         ...
8 }
9 __global__ void KernelDos( ... )
10 { ...
11 int main(...)
12 {
13     ...
14     L* datosD;
15     cudaMalloc(&datosD, size);
16     ...
17     cudaMemcpy(datosD, src, size, cudaMemcpyHostToDevice);
18     ...
19     dim3 gridDim(..., ..., ...);
20     dim3 blockDim(..., ..., ...);
21     KernelUno<<<gridDim, blockDim>>>(datosD, ... );
22     ...
23     KernelDos<<<..., ...>>>( ... );
24     ...
25     cudaMemcpy(res, datosD, size, cudaMemcpyHostToDevice);
26     ...
27     cudaFree(datosD);
28     ...
29 }
```

Código 3.3: Estructura general de un programa en CUDA C.

## CAPÍTULO 4

### SIFT EN GPU

Paralelizar correctamente un algoritmo no es una tarea trivial. Después del capítulo anterior al ver todas las ventajas que tenemos en las GPU, se puede decir que son la solución a todo. Tristemente, no lo son. Existen algoritmos que, por la estructura del programa y forma de ejecutar el proceso, no se pueden paralelizar. Para saber cómo analizar si un algoritmo es paralelizable, primero se dará una definición de qué es un programa paralelo:

*“Un programa paralelo es la especificación de dos o más procesos simultáneos que cooperan entre sí con un fin en común” [14]*

Se pueden destacar dos aspectos importantes de esta definición: el primero es la comunicación. Los procesos deben poder compartir información para poder trabajar simultáneamente sobre un mismo problema. El segundo es la sincronización, que es simplemente como organizar a los procesos para que realicen trabajo sin que interfieran al de otros procesos.

Entonces se debe cambiar la forma en que se hacen los programas, ahora no solo es el cómo llegar a un objetivo paso a paso, sino que además se tiene que pensar cómo muchos procesos trabajarán juntos para alcanzar un objetivo. Para lo que se debe dividir el algoritmo, los datos o los dos, para repartir el trabajo. Para paralelizar el algoritmo se analizan básicamente tres casos de paralelismo:

- *Funcional*: Lo que se divide es el algoritmo. Se buscan pasos en el algoritmo que no dependan de otra parte del mismo y se ejecutan simultáneamente en diferentes procesos. Requiere de sincronizar muy cuidadosamente, para que las diferentes partes del algoritmo no interfieran entre sí.

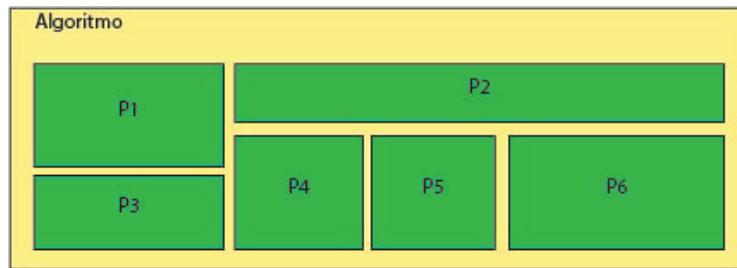


Figura 4-1: *Funcional*: Todos los procesos son partes diferentes del algoritmo

- *Dominio*: Se repartirán los datos en múltiples procesos los cuales tienen una especificación idéntica. La sincronización es sencilla en este caso, aun así hay que prestar atención ya que se podría corromper la información.

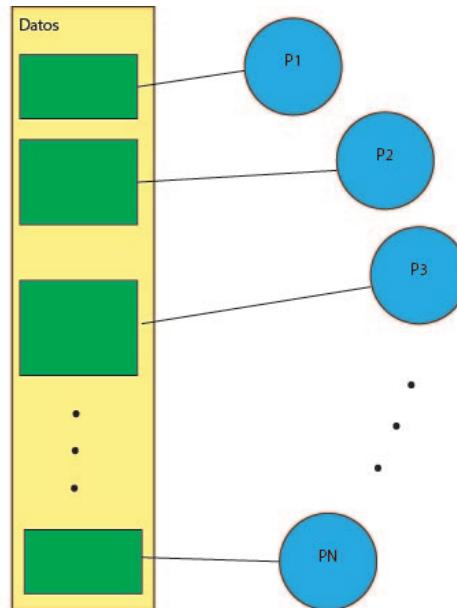


Figura 4-2: *Dominio*: Todos los procesos tienen la misma especificación

- *Actividad*: Es una combinación de los dos puntos anteriores.

Teniendo conocimiento sobre el algoritmo y las herramientas para mejorar su rendimiento

por medio de la paralelización, se analizaran las partes de SIFT para de esta manera adaptarlo al modelo de programación de CUDA.

### 4.1. Análisis de SIFT para su Paralelización en GPU

En esta sección se describe cómo se comunicarán y sincronizarán los procesos, así como la estructura que tomará el algoritmo de SIFT para poder paralelizarlo con CUDA.

Primero se dividirá en 6 partes el algoritmo de SIFT, como se muestra en la figura 4-3, para facilitar la programación en diferentes kernels los cuales no serán ejecutados simultáneamente.

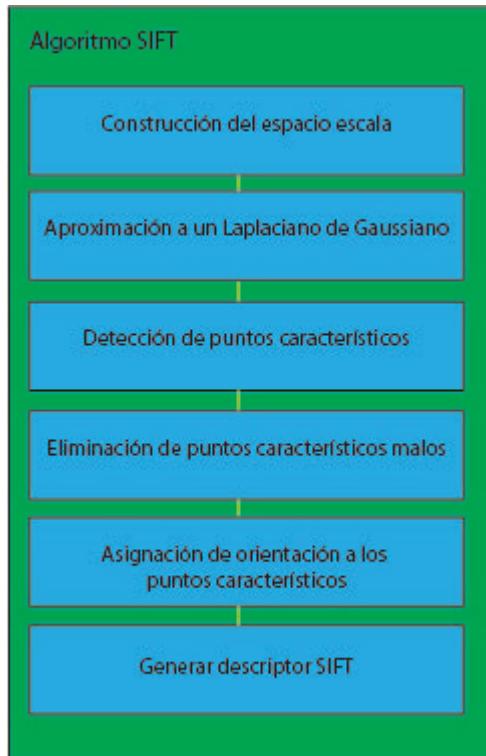


Figura 4-3: División del algoritmo SIFT a paralelizar

Los kernels de las diferentes partes del algoritmo tienen una estructura en común: todos tienen como entrada una o más imágenes (datos de solo lectura) y tienen como salida una imagen (o varias). Cada proceso tiene una sección de la imagen de tamaño  $N \times N$ , la cual puede estar traslapada con la de algún otro proceso. Sin embargo esta área no requiere de

#### 4.1. ANÁLISIS DE SIFT PARA SU PARALELIZACIÓN EN GPU

---

sincronización entre procesos ya que solo se usa para obtener datos no para procesarlos. En la imagen de salida al proceso, se le será asignado un solo pixel de la imagen. Como se puede mostrar en la figura 4-4, las zonas P1 y P2 están traslapadas en la imagen de salida (como si tuviera un zoom a los pixeles) no escriben en otro que no sea su pixel.

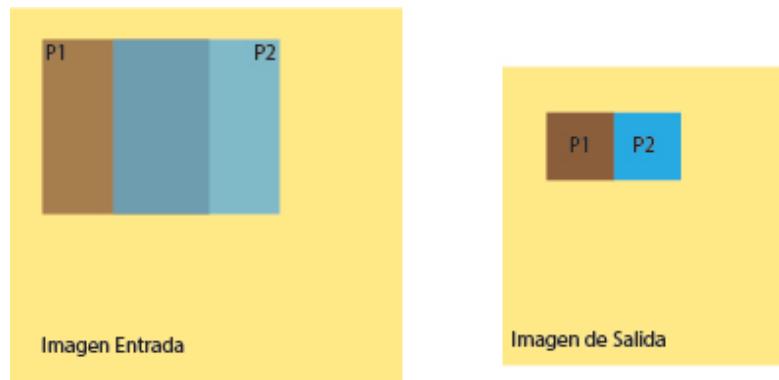


Figura 4-4: Proceso general de los kenels

Los procesos que se ejecutan sobre la imagen tienen la misma especificación. Es decir que lo que estamos repartiendo entre los múltiples procesos son los datos de entrada, con lo cual estaremos en la categoría de paralelismo de *dominio*.

Cada uno de los kernels será ejecutado múltiples veces. Este trabajo será desempeñado por el anfitrión (CPU) en forma secuencial. Esto es importante ya que cada uno de estos kernels es lanzado sin importar que el anterior terminara de ejecutarse. Si múltiples kernels son lanzados y tienen la misma especificación pero trabajan con diferentes secciones de los datos no existe problema. Pero si el anfitrión llegara a lanzar un kernel que tiene una especificación diferente a la de un banco de kernels lanzados anteriormente, y estos no han finalizado puede existir riesgo de corromper los datos. Entonces debemos sincronizar, como se puede ver en la figura 4-5, al dispositivo (GPU) con el anfitrión (CPU) para evitar caer en este tipo de errores.

## 4.2. IMPLEMENTACIÓN

---

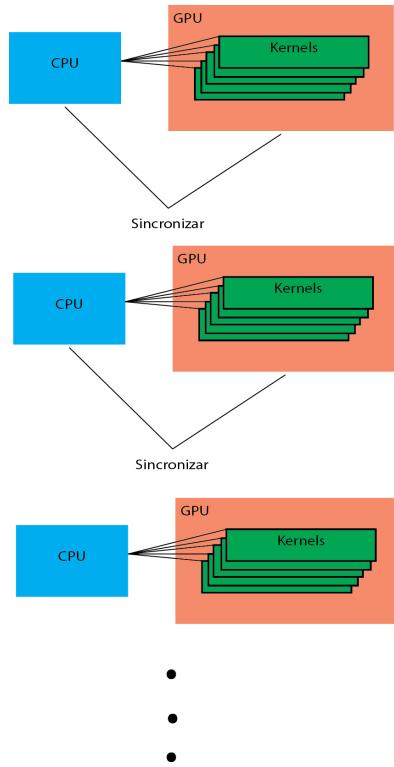


Figura 4-5: Lanzamiento de Kernels

## 4.2. Implementación

Teniendo en cuenta la estructura general de la solución, se plantea como se construyó y como es el funcionamiento de cada una de las partes en las que se dividió el algoritmo de SIFT (figura 4-3), para poder adaptarlo al modelo de programación de CUDA.



Figura 4-6: Esta es la imagen de entrada que arrojaron como resultado las de las figuras 4-8, 4-9, y 4-11

### 4.2.1. Construcción del espacio escala y aproximación a un Laplaciano de Gaussiana

Para construir el espacio escala con aproximaciones a un Laplaciano de Gaussiana se usaron diferentes filtros de diferencia de Gaussiana por cada escala, convolucionados con las imágenes de la escala.

$$D(x, y; \sigma) = (G(x, y; k\sigma) - G(x, y; \sigma)) * I(x, y)$$

Para ello se obtienen los filtros Gaussianos con la ayuda de la librería OpenCV [15], y se restan para así obtener los filtros que aplicados en cada octava, quedando como los de la figura 4-7.

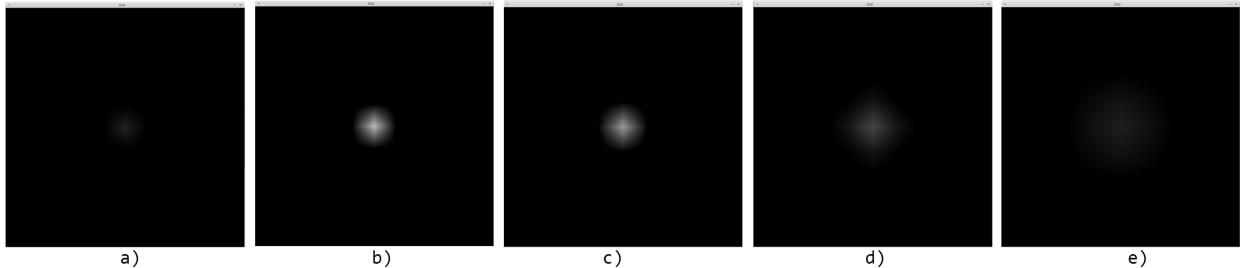


Figura 4-7: Filtros de diferencias de gaussianas. a)  $\sigma_0 = 1 - \sigma_1 = 1.08$ , b)  $\sigma_1 = 1.08 - \sigma_2 = 1.36$ , c)  $\sigma_2 = 1.36 - \sigma_3 = 1.72$ , d)  $\sigma_3 = 1.72 - \sigma_4 = 2.16$ , e)  $\sigma_4 = 2.16 - \sigma_5 = 2.72$ .

También se usó OpenCV para cambiar el tamaño de las imágenes, para cada octava. En lo que refiere a la construcción del espacio escala la parte que se paralelizo fue la convolución de una imagen y un filtro. Se puede ver como se hizo la implementación en el apéndice A. La idea es que estos kernels sean lanzados por el anfitrión. Se lanzarán tantos kernels como imágenes se necesitan para crear el espacio escala. Los kernels serán ejecutados de forma concurrente.

La función de cada kernel es, para cada pixel en la imagen de salida, asignar un hilo y estos se encargarán con los datos de entrada (la imagen y el filtro) de implementar la operación de convolución.

## 4.2. IMPLEMENTACIÓN

---

**Data:** ImgEntrada, Filtro

**Result:** Img

Para cada pixel en ImgEntrada asigna un hilo;

**forall hilos do**

```
    if Img.pixel es orilla then
        | Img.pixel=0;
    else
        | Img.pixel= ImgEntrada.zona * Filtro; // Donde * es el operador para la
          convolución
    end
```

**end**

**Algoritmo 1:** Cálculo de la convolución para cada imagen del espacio escala

El resultado se ejemplifica en figura 4-8, Tendremos imágenes muy similares para cada una de las octavas.



Figura 4-8: Espacio Escala de Diferencia de Gaussiana

### 4.2.2. Detección de puntos característicos

En el espacio escala obtenido anteriormente se buscarán los puntos extremos. En  $D(x, y, \sigma)$  se buscarán las ubicaciones máximas y mínimas. Cada punto es comparado con sus ocho vecinos en la misma imagen y con sus otros dieciocho vecinos de escala, nueve en la imagen de arriba y nueve en la imagen de abajo. Sólo se selecciona la ubicación si el pixel tiene un valor mayor o menor a todos sus vecinos.

**Data:** ImgArriba, Img , ImgAbajo

**Result:** ImgMascara

Para cada pixel en Img asigna un hilo;

**forall hilos do**

**if** *Img.pixel* es orilla **then**

ImgMascara.pixel=0;

**else**

**forall** *pixel vecino a Img en ImgArriba, ImgAbajo e Img* **do**

Compara cada pixel vecino con el pixel asignado al hilo;

**if** si todos los vecinos son menores o mayores **then**

ImgMascara.pixel = 1;

**else**

ImgMascara.pixel = 0;

**end**

**end**

**end**

**end**

**Algoritmo 2:** Búsqueda de puntos extremos

Como se puede ver en el algoritmo 2, lo que hacemos en el kernel es asignar a cada hilo un pixel de la imagen de entrada *Img* para compararla con sus vecinos y así poder determinar si es un punto extremo.

La imagen de salida terminará siendo una máscara binaria. Si existe un punto extremo, pondrá un pixel en blanco, y donde no, lo dejará en negro tal como se muestra en la figura 4-9; Se puede encontrar la implementación de este algoritmo en el apéndice B.

## 4.2. IMPLEMENTACIÓN

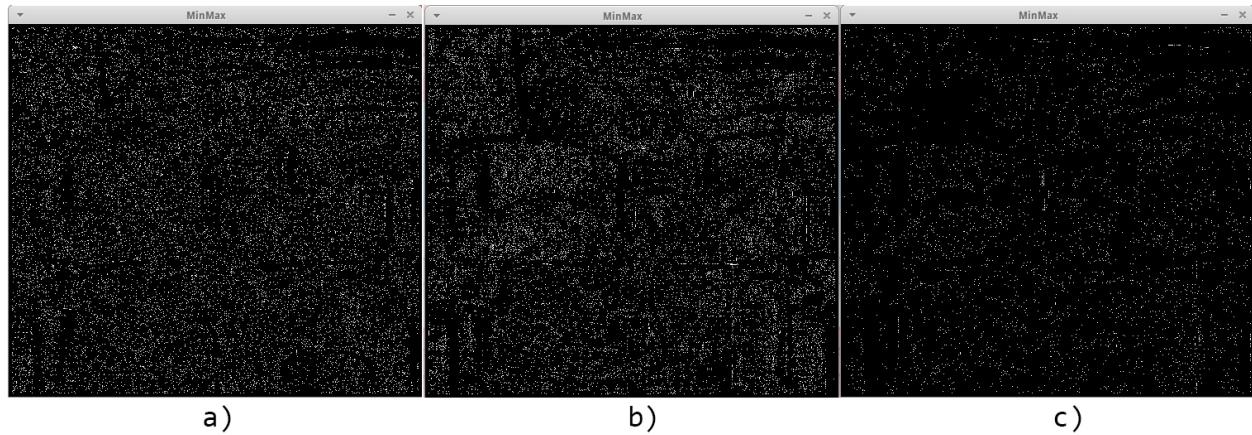


Figura 4-9: Mascara. Búsqueda con las imágenes del espacio escala: a) 0,1,2 ; b) 1,2,3; c)2,3,4

### 4.2.3. Eliminación de puntos característicos malos

A continuación se filtran los puntos extremos encontrados anteriormente. Existen dos casos donde los puntos extremos anteriormente seleccionados deben de ser eliminados, el primero es donde el contraste es muy bajo y el segundo es cuando se localiza en un borde.

**Data:** Img , ImgMascara

**Result:** ImgMascara

Para cada pixel en Img asigna un hilo;

**forall** hilos **do**

**if** ImgMascara.pixel es mayor que 0 **then**

**if** Img.pixel tiene contraste bajo o está en un borde **then**

            | ImgMascara.pixel=0;

**end**

**end**

**end**

**Algoritmo 3:** Eliminación de puntos característicos malos

Se obtendrá una máscara muy parecida a la de la Figura 4-10, pero esta vez se tienen muchos menos pixeles en color blanco. En el apéndice C se puede ver más detalladamente como se realizó la implementación de esta parte del algoritmo de SIFT.

## 4.2. IMPLEMENTACIÓN

---

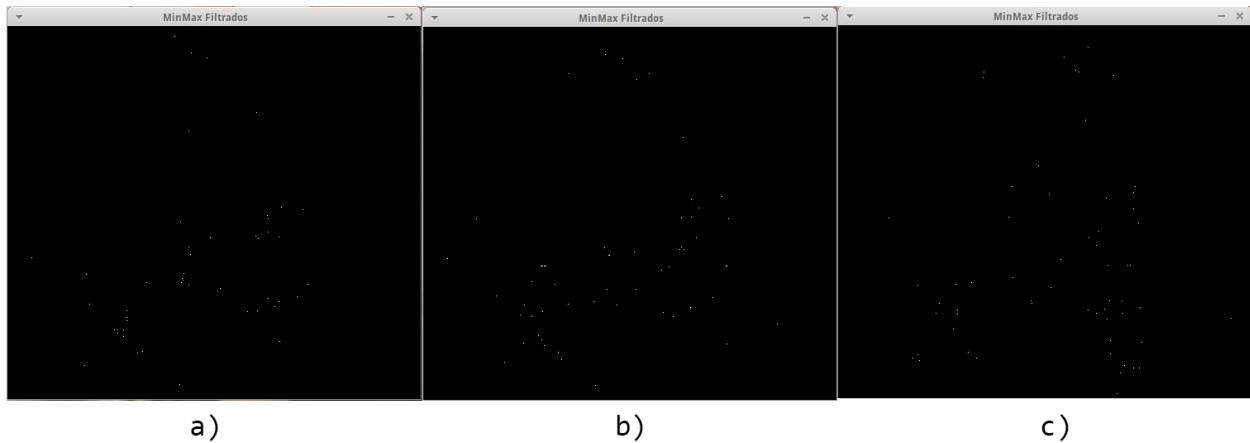


Figura 4-10: Máscara Filtrada de la búsqueda con las imágenes del espacio escala: a) 0,1,2 ; b) 1,2,3; c)2,3,4

### 4.2.4. Asignación de orientación a los puntos característicos

Encontrar la orientación de cada punto característico, basado en propiedades locales de la imagen, es importante para que el descriptor sea invariante a la rotación.

**Data:** Img

**Result:** ImgMagnitud, ImgOrientacion

Para cada pixel en Img asigna un hilo;

**forall hilos do**

Calcular magnitud en Img.pixel;

Calcular orientación en Img.pixel;

ImgMagnitud.pixel= magnitud;

ImgOrientacion.pixel=orientación;

**end**

**Algoritmo 4:** Cálculo de orientaciones y magnitudes en cada pixel

Para esta parte lo que se hizo fue dividir en dos kernels el proceso: uno para calcular las magnitudes y orientaciones de los gradientes. (Véase apéndice D)

$$m(x, y) = \sqrt{(L(x + 1, y) - L(x - 1, y))^2 + (L(x, y + 1) - L(x, y - 1))^2}$$

$$\theta(x, y) = \tan^{-1} \left( \frac{L(x, y + 1) - L(x, y - 1)}{L(x + 1, y) - L(x - 1, y)} \right)$$

## 4.2. IMPLEMENTACIÓN

Y el otro, donde se obtienen los histogramas para cada punto característico y se asigna una orientación dominante. (Véase apéndice E)

**Data:** Img , ImgMascara, ImgMagnitud, ImgOrientacion

**Result:** PuntosCaracteristicos

Para cada pixel en Img asigna un hilo;

**forall hilos do**

**if** *ImgMascara.pixel es mayor que 0 then*

    Realizar el histograma de la zona alrededor del punto característico;

    Encontrar cuales son los valores más altos más altos;

    Encontrar una orientación dominante con los puntos más altos;

    Almacenar la orientación y la ubicación de ese punto característico;

**end**

**end**

**Algoritmo 5:** Asignación de orientación respecto a su histograma

## 4.2. IMPLEMENTACIÓN

---

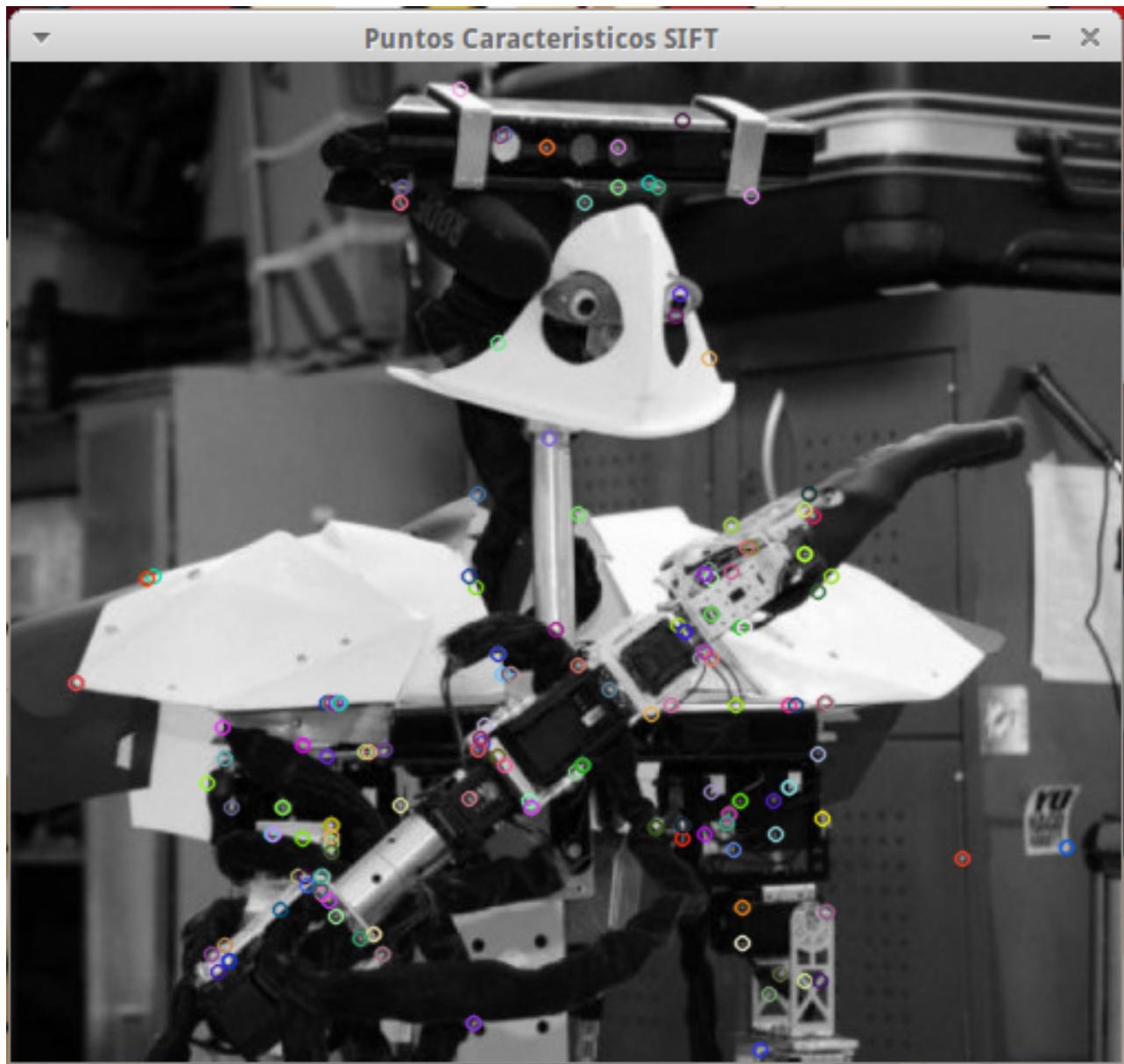


Figura 4-11: Puntos característicos encontrados

## CAPÍTULO 5

---

### PRUEBAS Y RESULTADOS

#### 5.1. Pruebas

Para las pruebas realizadas se utilizó una computadora con un procesador AMD Phenom II 720 con tres núcleos a 2.80Ghz cada núcleo, 10 GB de memoria RAM y una tarjeta gráfica NVIDIA GeForce GTX 650 Ti tiene una arquitectura Kepler con 768 núcleos CUDA, memoria de 2 GB y un ancho de banda para 86.4 GB/s. En cuanto al software las pruebas de hicieron bajo un sistema operativo xubuntu 14.04, utilizando opencv y CUDA 6.5.

Teniendo en cuenta el hardware y la forma en que se diseñaron los kernels se deben realizar ciertas optimizaciones. Para esta implementación como no se usa memoria compartida le daremos preferencia a la memoria cache L1 usando, la función **cudaFuncSetCacheConfig()** recibe 2 parámetros el primero será el nombre de la función del dispositivo (kernel) y el segundo es la configuración que se le dará a la memoria, en este caso *cudaFuncCachePreferL1*. Además, se debe tomar en cuenta la ocupación de los SM definida por la relación entre los *warps* activos y la cantidad de *warps* por SM. NVIDIA desarrollo una herramienta llamada CUDA Occupancy Calculator[16], la cual auxilia al desarrollador a encontrar la máxima ocupación para el lanzamiento de un kernel. Necesitará como datos de entrada el número de hilos por bloque, la cantidad de memoria compartida usada y el número de registros por hilo.

## 5.1. PRUEBAS

Otra herramienta que resulta muy útil para identificar problemas de rendimiento es el perfilador visual de NVIDIA[17], gracias a este se pudo analizar la ocupación teórica calculada contra la real para cada ejecución de los diferentes kernel lanzados con diferentes imágenes de entrada, como se puede ver en la tabla 5-1.

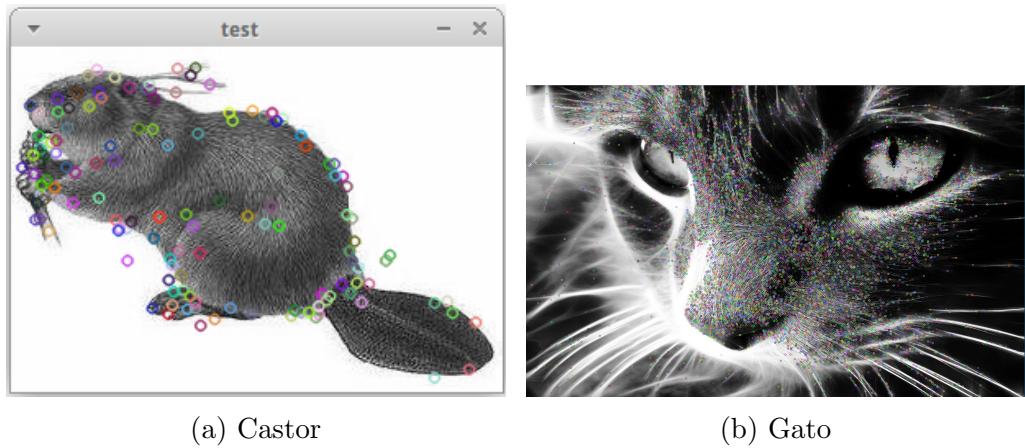
Ocupación de los SM			
Kernel	Teórica	Máxima Real	Mínima Real
Convolución	100 %	99 %	13 %
Localización de min-max	100 %	93 %	12 %
Remover puntos malos	100 %	93 %	28 %
Asignar magnitud y orientación	100 %	84 %	32 %
Puntos característicos	56 %	55 %	1.7 %

Tabla 5-1: Ocupación teórica vs ocupación real

Una vez dicho como se encontró la mejor condición de lanzamiento y distribución de la memoria para cada kernel, se puede describir como se realizaron las pruebas en general. Las pruebas se realizaron tomando un grupo de imágenes de diferentes resoluciones como entrada de la implementación propuesta, para medir el tiempo que le tomaba procesar la imagen y compararlo con otras implementaciones del algoritmo SIFT, ya existentes. La razón de porque imágenes de distintas resoluciones es por la manera tan diversa de obtener imágenes en el robot Justina.

## 5.2. Resultados

Hay dos imágenes que fueron las que ayudaron a realizar paso a paso el desarrollo de la implementación propuesta, la primera imagen es un castor, y la segunda imagen de un gato (figura 5-1). La razón por la cual se tomaron estas dos imágenes fue porque la imagen del castor es una imagen pequeña con pocos puntos característicos, y la del gato siendo de una resolución mayor y con una gran cantidad de puntos característicos. Siendo estas dos imágenes los extremos en cuanto a los datos de entrada.



(a) Castor

(b) Gato

Figura 5-1: Puntos característicos encontrados

La primer prueba que se hizo con el castor y el gato, consistió en medir el tiempo que tardaban en ejecutarse ciertas secciones del código de una implementación abierta de SIFT llamada Open SIFT [18], la cual se usa para la detección de objetos en el robot Justina, y en la implementación de SIFT con CUDA realizada para obtener los puntos característicos. Se puede observar en las tablas 5-2 y 5-3 los resultados de estas pruebas.

## 5.2. RESULTADOS

---

Castor		
Partes de SIFT	CUDA SIFT	Open SIFT
Espacio escala DoG	17.25 ms	29.32 ms
Detección y filtrado de PC	2.62 ms	50.09 ms
Orientación de PC	8.80 ms	12.35 ms

Tabla 5-2: La resolución de la imagen es de 300x211 px y se encontraron 120 puntos característicos

Gato		
Partes de SIFT	CUDA SIFT	Open SIFT
Espacio escala DoG	473.33 ms	957.19 ms
Detección y filtrado de PC	65.29 ms	2210.82 ms
Orientación de PC	125.2 ms	1014.89 ms

Tabla 5-3: La resolución de la imagen es de 1920x1200 px y se encontraron 12000 puntos característicos

Lo que se hizo después, fue medir el tiempo para obtener los puntos característicos en 3 diferentes implementaciones, la desarrollada para CUDA, la de OpenSIFT y por último la que se encuentra en OpenCV. Se pueden ver los resultados y el desempeño que se obtuvo en la tabla 5-4.

Castor				
Resolución	CUDA SIFT	Open SIFT	Opencv SIFT	Puntos Característicos
320 x 240 px	31.87 ms	93.22 ms	49.42 ms	120
Gato				
Resolución	CUDA SIFT	Open SIFT	Opencv SIFT	Puntos Característicos
1920x1200 px	676.32 ms	4221.41 ms	1415.98 ms	12000

Tabla 5-4: Tiempo de ejecucion de la implementacion en paralelo y 2 mas de forma secuencial

## 5.2. RESULTADOS

---

Castor		
Resolución	Open SIFT/CUDA SIFT	Opencv SIFT/CUDA SIFT
320 x 240 px	2.92	1.55
Gato		
Resolución	Open SIFT/CUDA SIFT	Opencv SIFT/CUDA SIFT
1920x1200 px	6.24	2.09

Tabla 5-5: Speedup entre la implementacion en paralelo y 2 mas de forma secuencial

Estas imágenes no eran las más adecuadas para hacer pruebas, ya que el robot de servicio Justina trabaja con imágenes como las de la figuras 5-3. Las primeras tres son objetos que tiene que manipular.



Figura 5-2: Puntos característicos encontrados

## 5.2. RESULTADOS

---

Lo que se hizo fue medir cuánto tiempo se tardaban en generar los puntos característicos en las 3 implementaciones anteriormente mencionadas y cambiar las resoluciones de estas imágenes, porque el robot no siempre vera los objetos del mismo tamaño. Dependerá de la cámara que se esté usando o que tan lejos esté viendo los objetos. Los resultados los podemos ver a continuación en las siguientes tablas:

Stevia				
Resolución	CUDA SIFT	Open SIFT	Opencv SIFT	Puntos Característicos
320 x 240 px	31.87 ms	151.10 ms	49.42 ms	370
640 x 480 px	97.91 ms	484.64 ms	182.46 ms	920
1280 x 960 px	335.05 ms	1751.10 ms	679.60 ms	2800
2560 x 1920 px	1251.38 ms	5911.26 ms	2893.68 ms	3000

Tabla 5-6: Tiempo de ejecución de la implementación en paralelo y 2 más de forma secuencial

Stevia		
Resolución	Open SIFT/CUDA SIFT	Opencv SIFT/CUDA SIFT
320 x 240 px	4.74	1.55
640 x 480 px	4.94	1.86
1280 x 960 px	5.22	2.02
2560 x 1920 px	4.72	2.31

Tabla 5-7: Speedup la implementación en paralelo y 2 más de forma secuencial

Café				
Resolución	CUDA SIFT	Open SIFT	Opencv SIFT	Puntos Característicos
320 x 180 px	30.25 ms	117.95 ms	39.25 ms	290
640 x 360 px	81.91 ms	484.64 ms	182.46 ms	760
1280 x 720 px	284.57 ms	1415.27 ms	518.27 ms	2800
2560 x 1440 px	993.96 ms	4963.27 ms	1951.21 ms	7000

Tabla 5-8: Tiempo de ejecución de la implementación en paralelo y 2 más de forma secuencial

## 5.2. RESULTADOS

---

Cafe		
Resolución	Open SIFT/CUDA SIFT	Opencv SIFT/CUDA SIFT
320 x 240 px	3.89	1.29
640 x 480 px	5.91	2.22
1280 x 960 px	4.97	1.82
2560 x 1920 px	4.99	1.96

Tabla 5-9: Speedup la implementación en paralelo y 2 más de forma secuencial

Sopa				
Resolución	CUDA SIFT	Open SIFT	Opencv SIFT	Puntos Característicos
206 x 240 px	28.10 ms	130.98 ms	37.78 ms	425
411 x 480 px	74.43 ms	412.04 ms	129.49 ms	1000
802 x 906 px	241.83 ms	1400.59 ms	469.07 ms	3000
1645 x 1920 px	850.29 ms	4478.54 ms	1674.67 ms	3900

Tabla 5-10: Tiempo de ejecución de la implementación en paralelo y 2 más de forma secuencial

Sopa		
Resolución	Open SIFT/CUDA SIFT	Opencv SIFT/CUDA SIFT
320 x 240 px	4.66	1.34
640 x 480 px	5.54	1.73
1280 x 960 px	5.79	1.93
2560 x 1920 px	5.26	1.96

Tabla 5-11: Speedup la implementación en paralelo y 2 más de forma secuencial

Estante				
Resolución	CUDA SIFT	Open SIFT	Opencv SIFT	Puntos Característicos
320 x 240 px	33.45 ms	130.24 ms	47.52 ms	210
640 x 480 px	101.20 ms	451.39 ms	177.28 ms	700
1280 x 960 px	340.92 ms	1602.51 ms	669.09 ms	2100
2560 x 1920 px	1275.37 ms	6031.39 ms	3037.60 ms	3700

Tabla 5-12: Tiempo de ejecución de la implementación en paralelo y 2 más de forma secuencial

Estante		
Resolución	Open SIFT/CUDA SIFT	Opencv SIFT/CUDA SIFT
320 x 240 px	3.89	1.42
640 x 480 px	4.46	1.75
1280 x 960 px	4.70	1.96
2560 x 1920 px	4.73	2.38

Tabla 5-13: Speedup la implementación en paralelo y 2 más de forma secuencial

## *5.2. RESULTADOS*

---

Un factor que pudo afectar el tiempo de ejecución de todos los programas fue la cantidad de puntos característicos que existen en la imagen, cosa que no pasó. Se puede observar cómo se repite el fenómeno que notamos con el castor y el gato. Entre más grande es la imagen obtenemos un desempeño más grande, esto paso para todos los casos en las imágenes anteriores. Es importante mencionar que el tiempo medido en las tablas incluye el cuello de botella que existe al estar pasando datos de la memoria RAM de la computadora a la memoria de la GPU.

## CAPÍTULO 6

---

### CONCLUSIONES Y TRABAJO A FUTURO

#### 6.1. Conclusiones

Las actividades que el robot de servicio Justina desempeña son más rigurosas cada año, un ejemplo de esto es una nueva prueba propuesta en Robocup 2015 llamada *Manipulation and object recognition*, donde se tiene que buscar objetos en un librero. Para esta prueba solo se cuenta con 3 minutos para reconocer y manipular objetos, de los cuales casi todo el tiempo se invierte en reconocimiento, ya que con una sola foto no podríamos analizar todos los objetos que hay en el librero, por lo cual, hay que hacer varias tomas de diferente ángulos. Para el reconocimiento de objetos el equipo de Biorobotica de la UNAM usa la siguiente estrategia: se segmentan los objetos y se procesa solo una cantidad pequeña de pixels para el reconocimiento, usando un sensor Kinect para obtener las imágenes, este tiene una resolución de 640 x 480 px, y se utiliza el código de openSIFT para procesar estas imágenes. Desafortunadamente el intento por segmentar los objetos de un librero no han sido existo, por lo que hay que analizar imágenes más grandes y desde una mayor distancia. La implicación de esto es la necesidad de mayor resolución en las imágenes para no perder detalle de los objetos y tener una manera más rápida de procesar estas nuevas imágenes. Por esto se adquirió una cámara RGB Logitech C920 que tiene una resolución de 1920 x 1080 px, y el procesamiento de la imagen fue mucho más lento pero mostro una mejora en cuanto a reconocimiento. Se planea empezar a usar también el nuevo sensor de Microsoft Kinect 2

el cual también tiene una resolución de 1920 x 1080 px.

El objetivo de este trabajo fue obtener una manera de procesar más rápido estas imágenes de alta resolución. Y podemos decir que el objetivo se cumplió, ya que obtuvimos una mejora en el desempeño, como se puede ver en el capítulo Pruebas y Resultados, ya que en promedio se obtuvo un speedup de 4.94, es un buen resultado en general. Aunque analizando los resultados encontraremos los mejores cuando la resolución de la imagen es de 1280 x 960 px, esto lo podemos relacionar con la ocupación de los SM en la tabla 5-1 encontramos un máximo y un mínimo, cuando una imagen es grande la ocupación estará en su máximo pero si la imagen es pequeña la ocupación no será la más óptima. Entonces, la manera más óptima de utilizar esta implementación de SIFT, en GPU, será utilizando imágenes de alta resolución para obtener los mejores tiempos sin perder detalles por no tener suficiente información.

## 6.2. Trabajo a Futuro

Como se vio, hay nuevas necesidades para el robot Justina, por lo que hay que seguir trabajando en esto. Se deben hacer pruebas en diferentes tarjetas gráficas más poderosas y de nueva generación. Pero no solo el hardware es lo que hay que probar, como vimos en el capítulo tres, los diferentes tipos de memoria ayudan a hacer más eficiente el acceso a los datos, experimentar con estos tipos de memoria para buscar un mejor desempeño.

Encontrar otra forma más rápida de pasar los datos de la memoria RAM a la memoria de la GPU (ese es un cuello de botella importante) y encontrar otra forma de manejar las imágenes para que los accesos a memoria sean más rápidos.

Algo que ayudaría a Justina, es desarrollar el algoritmo para que genere el descriptor de SIFT y otro para hacer el emparejamiento de estos descriptores en paralelo. Este último podría ser el más importante ya que una vez que se obtienen los descriptores, compararlos con todos los encontrados en una imagen es una tarea muy sencilla, pero ardua ya que la cantidad de descriptores que se pueden encontrar en una imagen de alta resolución es grande, y la cantidad de descriptores del objeto también es alta, haciendo el tiempo de esta tarea muy alto.

## APÉNDICE A

---

### KERNEL CONVOLUCIÓN

```
1 --global__ void Convolution(float* image, float* mask,
2     ArrayImage* PyDoG, int maskR, int maskC,
3     int imgR, int imgC, float* imgOut, int idxPyDoG)
4 {
5     int tid= threadIdx.x;
6     int bid= blockIdx.x;
7     int bDim=blockDim.x;
8     int gDim=gridDim.x;
9     int iImg=0;
10    float aux=0;
11    int pxlThrd = ceil((double)(imgC*imgR)/(gDim*bDim));
12    for(int i = 0; i < pxlThrd; ++i)
13    {
14        iImg=(tid+(bDim*bid)) + (i*gDim*bDim);
15        if (iImg < imgC*imgR)
16        {
17            int condition=maskC/2+imgC*(floor((double)maskC/2));
18            if (iImg-condition < 0 ||
19                iImg+condition > imgC*imgR ||
20                iImg%imgC < maskC/2 ||
21                iImg%imgC > (imgC-1)-(maskC/2) )
22            {
23                aux=0;
24            }else
25            {
26                int itMask = 0;
27                int itImg=iImg-condition;
28                for (int j = 0; j < maskR; ++j)
```

---

```
29         {
30             for (int h = 0; h < maskC; ++h)
31             {
32                 aux+=image[itImg]*mask[itMask];
33                 ++itMask;
34                 ++itImg;
35             }
36             itImg+=imgC-maskC;
37         }
38     }
39     imgOut[iImg]=aux;
40     aux=0;
41 }
42 }
43 PyDoG[idxPyDoG].image=imgOut;
44 }
```

## APÉNDICE B

---

### KERNEL LOCALIZACIÓN DE MÁXIMOS Y MÍNIMOS

```
1 --global__ void LocateMaxMin(ArrayImage* PyDoG, int idxPyDoG ,
2     float * imgOut ,MinMax * mM, int maskC, int imgR ,
3     int imgC, int idxmM)
4 {
5     int tid= threadIdx.x;
6     int bid= blockIdx.x;
7     int bDim=blockDim.x;
8     int gDim=gridDim.x;
9     int iImg=0;
10    int pxlThrd = ceil((double)(imgC*imgR)/(gDim*bDim));
11    for(int i = 0; i <pxlThrd; ++i)
12    {
13        int min=0;
14        int max=0;
15        float value=0.0;
16        float compare =0.0;
17        iImg=(tid+(bDim*bid)) + (i*gDim*bDim);
18        if(iImg < imgC*imgR)
19        {
20            int condition=maskC/2+imgC*(floor((double)maskC/2));
21            if (iImg-condition < 0 || 
22                iImg+condition > imgC*imgR ||
23                iImg%imgC < maskC/2 ||
24                iImg%imgC > (imgC-1)-(maskC/2) )
25            {
26                imgOut[iImg]=0;
27            }else
28            {
29                value=PyDoG[idxPyDoG].image[iImg];
30                for (int m = -1; m < 2; ++m)
31                {
```

---

```
32         int itImg=iImg-(1+imgC);
33         for (int j = 0; j < 3; ++j)
34         {
35             for (int h = 0; h < 3; ++h)
36             {
37                 compare =PyDoG[idxPyDoG+m].image[itImg];
38                 if(value<=compare && max==0)
39                 {
40                     ++min;
41                 }
42                 else if(value>=compare && min==0)
43                 {
44                     ++max;
45                 }
46                 ++itImg;
47             }
48             itImg+=imgC-3;
49         }
50     }
51     if( (min==26 || max==26)) {
52         imgOut[iImg]=1;
53     }else
54     {
55         imgOut[iImg]=0;
56     }
57 }
58 }
59 }
60 mM[idxmM].minMax=imgOut;
61 }
```

## APÉNDICE C

### KERNEL REMOVER PUNTOS MALOS

```
1  __global__ void RemoveOutlier(ArrayImage* PyDoG, MinMax * mM,
2      int idxmM, int idxPyDoG, int imgR,int imgC ,float* auxOut)
3  {
4      int tid= threadIdx.x;
5      int bid= blockIdx.x;
6      int bDim=blockDim.x;
7      int gDim=gridDim.x;
8      int iImg=0;
9      int pxlThrd = ceil((double)(imgC*imgR)/(gDim*bDim));
10     for(int i = 0; i <pxlThrd; ++i
11     {
12         iImg=(tid+(bDim*bid)) + (i*gDim*bDim);
13         if(iImg < imgC*imgR)
14         {
15             if (mM[idxmM].minMax[iImg]>0 &&
16                 PyDoG[idxPyDoG].image[iImg]>0.05)
17             {
18                 float d, dxx, dyy, dxy, tr, det;
19                 d = PyDoG[idxPyDoG].image[iImg];
20                 dxx = PyDoG[idxPyDoG].image[iImg+1] +
21                     PyDoG[idxPyDoG].image[iImg-1] - (2*d);
22                 dyy = PyDoG[idxPyDoG].image[iImg+imgC] +
23                     PyDoG[idxPyDoG].image[iImg-imgC] - (2*d);
24                 dxy = (PyDoG[idxPyDoG].image[iImg+1+imgC] -
25                     PyDoG[idxPyDoG].image[iImg-1+imgC] -
26                     PyDoG[idxPyDoG].image[iImg+1-imgC] +
27                     PyDoG[idxPyDoG].image[iImg-1-imgC])/4.0;
28                 tr = dxx + dyy;
29                 det = dxx*dyy - dxy*dxy;
30                 if(det<=0 && !(tr*tr/det < 12.1))
31                     mM[idxmM].minMax[iImg]=0;
```

---

```
32     }else
33     {
34         mM[idxmM].minMax[iImg]=0;
35     }
36     auxOut[iImg]=mM[idxmM].minMax[iImg];
37 }
38 }
39 }
```

## APÉNDICE D

---

### KERNEL ASIGNAR MAGNITUD Y ORIENTACIÓN

```
1 --global__ void OriMag(ArrayImage* PyDoG, int idxPyDoG,
2     int imgR,int imgC , ArrayImage* Mag, ArrayImage* Ori,
3     int idxMagOri, float* MagAux, float* OriAux)
4 {
5     int tid= threadIdx.x;
6     int bid= blockIdx.x;
7     int bDim=blockDim.x;
8     int gDim=gridDim.x;
9     float dx,dy;
10    int iImg=0;
11    int pxlThrd = ceil((double)(imgC*imgR)/(gDim*bDim));
12    for(int i = 0; i <pxlThrd; ++i)
13    {
14        iImg=(tid+(bDim*bid)) + (i*gDim*bDim);
15        if(iImg < imgC*imgR)
16        {
17            int condition=1/2+imgC*(floor((double)1/2));
18            if (iImg-condition < 0 ||
19                iImg+condition > imgC*imgR ||
20                iImg%imgC < 1/2 ||
21                iImg%imgC > (imgC-1)-(1/2) )
22            {
23                OriAux [iImg]=0;
24                MagAux [iImg]=0;
25            }else
26            {
27                dx=PyDoG[idxPyDoG].image[iImg+1]-
28                    PyDoG[idxPyDoG].image[iImg-1];
29                dy=PyDoG[idxPyDoG].image[iImg+imgC]-
30                    PyDoG[idxPyDoG].image[iImg-imgC];
31                MagAux [iImg]=sqrt(dx*dx + dy*dy);
```

---

```
32         OriAux[iImg]=atan2(dy,dx);
33     }
34 }
35 }
36 Mag[idxMagOri].image= MagAux;
37 Ori[idxMagOri].image= OriAux;
38 }
```

## APÉNDICE E

---

### KERNEL PUNTOS CARACTRISTICOS

```
1  __global__ void KeyPoints(ArrayImage * Mag, ArrayImage * Ori,
2                          MinMax * mM, int idxM0mM, keyPoint * KP,
3                          float sigma, int imgR, int imgC, int octava )
4  {
5      int tid= threadIdx.x;
6      int bid= blockIdx.x;
7      int bDim=blockDim.x;
8      int gDim=gridDim.x;
9      float o = 0, val=0;
10     int x=0, y=0, octv=-1;
11     int iImg=0;
12     int pxlThrd = ceil((double)(imgC*imgR)/(gDim*bDim));
13     for(int i = 0; i <pxlThrd; ++i)
14     {
15         iImg=(tid+(bDim*bid)) + (i*gDim*bDim);
16         octv=-1;
17         if(iImg < imgC*imgR )
18         {
19             if(mM[idxM0mM].minMax[iImg]>0 )
20             {
21                 float histo[36]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
22                               0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
23                 octv=octava;
24                 x=iImg %imgC;
25                 y=iImg /imgC;
26                 int idxM0= (iImg -5)-(5*imgC);
27                 float exp_denom = 2.0 * sigma * sigma;
28                 float w;
29                 int bin;
30                 for (int i = -5; i < 6; ++i)
31                 {
```

```

32         for (int j = -5; j < 6; ++j)
33     {
34         w = exp( -( i*i + j*j ) / exp_denom );
35         bin =(Ori[idxM0mM].image[idxM0]<0)?
36             (18*(6.283185307-Ori[idxM0mM].image[idxM0])/3.141592654):
37             (18*Ori[idxM0mM].image[idxM0]/3.141592654);
38         histo[bin]+= w*Mag[idxM0mM].image[idxM0];
39         ++idxM0;
40     }
41     idxM0=idxM0+imgC-11;
42 }
43 int idxH=0;
44 float valMaxH = histo[0];
45 for (int i = 1; i < 36; ++i)
46 {
47     if(histo[i]>valMaxH){
48         idxH = i;
49         valMaxH=histo[i];
50     }
51 }
52 int l = (idxH == 0)? 35:idxH-1;
53 int r = (idxH+1)%36;
54 float bin_;
55 bin_= idxH + ((0.5*(histo[l]-histo[r]))/
56                 (histo[l]-(2*histo[idxH])+histo[r]));
57 bin_= ( bin_ < 0 )? 36 + bin_ :
58     ( bin_ >= 36 )? bin_ - 36 : bin_;
59 o=((360*bin_)/36);
60 val=valMaxH;
61 }else
62 {
63     o=-1.0;
64     x=-1;
65     y=-1;
66     octv=-1;
67 }
68 KP[iImg].orientacion=o;
69 KP[iImg].x=x;
70 KP[iImg].y=y;
71 KP[iImg].octv=octv;
72 KP[iImg].size=val;
73 }
74 }
75 }
```

## BIBLIOGRAFÍA

- [1] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, November 2004.
- [2] C. Harris and M. Stephens. A Combined Corner and Edge Detector. *Proceedings of the Alvey Vision Conference 1988*, pages 147–151, 1988.
- [3] David Kirk and Wen-mei Hwu. *Programming massively parallel processors : a hands-on approach*. Morgan Kaufmann, San Francisco, CA, USA, 2010.
- [4] NVIDIA. Cuda c programming guide [en linea]. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3yegx1PRY>, Accedido: [25/08/2015].
- [5] Jack Dongarra. Lo que la gente dice [en linea]. [http://www.nvidia.com.mx/object/cuda\\_home\\_new\\_la.html](http://www.nvidia.com.mx/object/cuda_home_new_la.html), Accedido: [22/08/2015].
- [6] NVIDIA. The world’s first gpu geforce 256 [en linea]. <http://www.nvidia.es/page/geforce256.html>, Accedido: [25/08/2015].
- [7] NVIDIA. Gpu-accelerated applications [en linea]. <http://la.nvidia.com/content/gpu-applications/PDF/gpu-applications-catalog.pdf>, Accedido: [04/05/2016].
- [8] I Buck. Brook : A Streaming Programming Language. *Communication*, (October):1–12, 2001.

- [9] NVIDIA. Language solutions [en linea]. <https://developer.nvidia.com/language-solutions>, Accedido: [25/08/2015].
- [10] Whitepaper Nvidia, Next Generation, and Cuda Compute. Whitepaper NVIDIA's Next Generation CUDA Compute Architecture. *ReVision*, 23(6):1–22, 2009.
- [11] Nvidia. Kepler GK110. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*, 2012.
- [12] Nvidia Geforce Gtx, Featuring Maxwell, The Most, Advanced Gpu, and Ever Made. NVIDIA GeForce GTX 980. pages 1–32.
- [13] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. EBL-Schweitzer. Wiley, 2014.
- [14] Jorge L. Ortega Arjona y José Galavíz Casas. *Programación Concurrente*. Maestría en Ciencias de la Computación, UACPyP del C.C.H., UNAM, 1996.
- [15] OpenCV. Opencv documentation [en linea]. <http://docs.opencv.org/2.4/index.html>, Accedido: [25/08/2015].
- [16] NVIDIA. Nvidia occupancy calculator [en linea]. [http://developer.download.nvidia.com/compute/cuda/CUDA\\_Occupancy\\_calculator.xls](http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls), Accedido: [05/05/2016].
- [17] NVIDIA. Nvidia visual profiler [en linea]. <https://developer.nvidia.com/nvidia-visual-profiler>, Accedido: [05/05/2016].
- [18] Rob Hess. Opensift [en linea]. <https://robwhess.github.io/opensift/l>, Accedido: [25/08/2015].
- [19] Utkarsh Sinha. Sift: Keypoint orientations [en linea]. <http://aishack.in/tutorials/sift-scale-invariant-feature-transform-keypoint-orientation/>, Accedido: [25/08/2015].
- [20] RAE. Diccionario de la lengua española [en linea]. <http://dle.rae.es/?w=diccionario>, Accedido: [04/05/2016].

## ÍNDICE DE FIGURAS

2-1. Espacio Escala de Diferencia de Gaussiana . . . . .	6
2-2. Espacio Escala de Diferencia de Gaussiana . . . . .	7
2-3. Histograma de Orientación . . . . .	10
2-4. Descriptor . . . . .	11
2-5. Operaciones en Punto Flotante por segundo de GPU y CPU [4] . . . . .	13
3-1. Sistema Híbrido con GPU y CPU . . . . .	14
3-2. SIMD . . . . .	17
3-3. La arquitectura Fermi tiene sus 16 SM alrededor de la memoria compartida L2 cache [10] Ver figura 3-4 para detalle de los SM. . . . .	18
3-4. Fermi Streaming Multiprocessor (SM)[10] . . . . .	19
3-5. Kepler Next Generation Streaming Multiprocessor (SMX) [11] . . . . .	20
3-6. Maxwell Streaming Multiprocessor (SMM) [12] . . . . .	22
3-7. Organización bloques en malla e hilos en bloques [4] . . . . .	24
3-8. Asignación de bloques por SM [4] . . . . .	25
3-9. Tipos de Memoria de la GPU [13] . . . . .	26
3-10. Programación Heterogénea [4] . . . . .	28
4-1. <i>Funcional</i> :Todos los procesos son partes diferentes del algoritmo . . . . .	31
4-2. <i>Dominio</i> :Todos los procesos tienen la misma especificación . . . . .	31

---

4-3. División del algoritmo SIFT a paralelizar . . . . .	32
4-4. Proceso general de los kenels . . . . .	33
4-5. Lanzamiento de Kernels . . . . .	34
4-6. Esta es la imagen de entrada que arrojaron como resultado las de las figuras 4-8, 4-9, y 4-11 . . . . .	34
4-7. Filtros de diferencias de gaussianas. a) $\sigma_0 = 1 - \sigma_1 = 1.08$ , b) $\sigma_1 = 1.08 - \sigma_2 = 1.36$ , c) $\sigma_2 = 1.36 - \sigma_3 = 1.72$ , d) $\sigma_3 = 1.72 - \sigma_4 = 2.16$ , e) $\sigma_4 = 2.16 - \sigma_5 = 2.72$ . . . . .	35
4-8. Espacio Escala de Diferencia de Gaussiana . . . . .	36
4-9. Mascara. Búsqueda con las imágenes del espacio escala: a) 0,1,2 ; b) 1,2,3; c) 2,3,4 . . . . .	38
4-10. Máscara Filtrada de la búsqueda con las imágenes del espacio escala: a) 0,1,2 ; b) 1,2,3; c) 2,3,4 . . . . .	39
4-11. Puntos característicos encontrados . . . . .	41
5-1. Puntos característicos encontrados . . . . .	44
5-2. Puntos característicos encontrados . . . . .	46