

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO



UNIVERSIDAD NACIONAL
AUTÓNOMA DE
MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN
LABORATORIO DE BIO-ROBÓTICA

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

1. Introducción	1
1.1. Contexto	1
1.2. Problema a resolver	2
1.3. Hipótesis	2
1.4. Estructura de la tesis	3
2. Marco Teórico	5
2.1. Extracción y descripción de características	5
2.1.1. Características invariantes a transformaciones afines	6
2.2. Computo en GPU	13
3. Unidad de Procesamiento de Graficos de Proposito General	16
3.1. Breve Historia	17
3.2. Plataforma y modelo de programación de computo paralelo	18
3.2.1. Arquitecturas	19
3.3. Modelo de Programación CUDA C	27
3.3.1. Kernels	27
3.3.2. Jerarquía de Hilos	27

3.3.3. Jerarquía de Memoria	30
3.3.4. Programación heterogénea	33
4. SIFT en GPU	37
4.1. Análisis de SIFT para su Paralelización en GPU	39
4.2. Implementación	41
4.2.1. Construcción del espacio escala y aproximación a un Laplaciano de Gaussiano	41
4.2.2. Detección de puntos característicos	45
4.2.3. Eliminación de puntos característicos malos	47
4.2.4. Asignación de orientación a los puntos característicos	48
5. Pruebas y Resultados	51
6. Conclusiones y Trabajo a Futuro	59
6.1. Conclusiones	59
6.2. Trabajo a Futuro	60
A. Kernel Convolución	62
B. Kernel Localización de máximos y mínimos	64
C. Kernel Remover puntos malos	67
D. Kernel Asignar magnitud y orientación	69
E. Kernel Generar puntos característicos	71
Bibliografía	74
Índice de figuras	76

CAPÍTULO 1

INTRODUCCIÓN

1.1. Contexto

Los robots móviles rigen su comportamiento con base en el software, dependiendo de cuanta interacción tenga el robot con su entorno, podría llegar a ser más complejo que el hardware que lo conforma. Una parte importante de este software, es la forma en la que el robot puede obtener datos para darles un significado e interpretarlos.

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, en la robótica, tenga una participación muy importante, porque estas máquinas empiezan a ser utilizadas, para tareas que antes solo los humanos realizaban. Entonces las 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, la forma de ganar tiempo que se ha venido sesgando por hardware es el paralelismo, y no se trata solo de procesadores multinúcleo, las unidades de procesamiento gráfico(GPU) se pueden programar para realizar tareas de propósito general.

1.2. Problema a resolver

Los algoritmos que se utilizan, en visión computacional la mayor parte de las veces son muy confiables, pero consumen mucho tiempo de procesador, por esto se ha tratado de hacer más eficientes estos algoritmos, pero provoca que la confiabilidad de estos disminuya. El tiempo en el cual se adquieren y procesa la información, es crucial en la actividad de un robot, de esto depende que decisión tomara.

Con base en lo anterior, lo importante es el tiempo en que procesemos los datos, para tomar una decisión, pero igual de importante es que la información obtenida sea congruente.

En muchos casos, el software que funciona en paralelo es más rápido que el secuencial, podemos ver que los algoritmos que se manejan en visión computacional son siempre secuenciales. Otro punto importante son los recursos, como procesador y 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 cuanto al tiempo en el que se ejecuta, de un algoritmo para encontrar puntos característicos que sean invariantes a transformaciones afines"

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 medirá comparando el tiempo que se obtiene, con el sistema actual del robot y el

que se propone.

1.4. Estructura de la tesis

1. Introducción. En este capítulo se presenta de lo que tratará en general este trabajo de tesis, planteando el contexto en el que se trabaja, el problema a resolver, la hipótesis y cuál sería el alcance de este trabajo.

2. Marco Teórico. Se verán 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 tolerables 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.

3. Unidad de Procesamiento de Gráficos de Propósito General . Este capítulo tratará de cómo han cambiado, los procesadores multinúcleo, la forma en la que programamos y sobre todo el cómputo en cooperación con las tarjetas gráficas. Nos enfocaremos en las GPU de la familia de Nvidia, se verá 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.

4. SIFT en GPU. Se presentaran puntos importantes al momento de paralelizar un algoritmo. Como es que se propone dividir el algoritmo de SIFT, para que sea más sencillo el proceso de paralelizarlo y como trabajan los kernels en general para este caso. Más adelante se explica como es que están estructuradas una a una las secciones, en las que dividimos el algoritmo.

5. Pruebas y Resultado. Se compara el tiempo que nos toma obtener los puntos característicos SIFT con el sistema propuesto y algunos otros ya implementados como la librería OpenSIFT y OpenCV.
6. Conclusiones y Trabajo a Futuro. Aun implementado el algoritmo para las GPU de Nvidia existe una gama muy variada, en general el siguiente paso es adaptar esta implementación al GPU de la computadora del robot para obtener un mejor desempeño.

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 discernir de los objetos que está en una imagen nos basamos en las características encontradas, 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 de la información del objeto, esto facilitara la comparación entre los diferentes objetos que existan en una imagen.

Para extraer las características existen diferentes formas, dependerá de que algoritmo se utilice, cada uno de ellos se enfoca en encontrar características como esquinas, bordes, crestas y regiones que son más obscuras o más claras; no todos los reconocedores encontraran las mismas o todos los tipos de características. Para que estos puntos sean robustos deben poder ser encontrados aun que los objetos se encuentren rotados, escalados, con cambios de iluminación o si están parcialmente ocultos.

A continuación, se explicara cómo se obtienen los puntos característicos por medio del algoritmo de SIFT (Scale-Invariant Feature Transform).

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 σ . Está representado por la función $L(x, y; \sigma)$ la cual se forma por la convolución con $G(x, y; \sigma)$ y 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 , 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 aplicará el espacio escala, usando diferencias de gaussianas convolucionadas con una imagen, en lugar de solo un filtro gaussiano, $D(x, y; \sigma)$ que

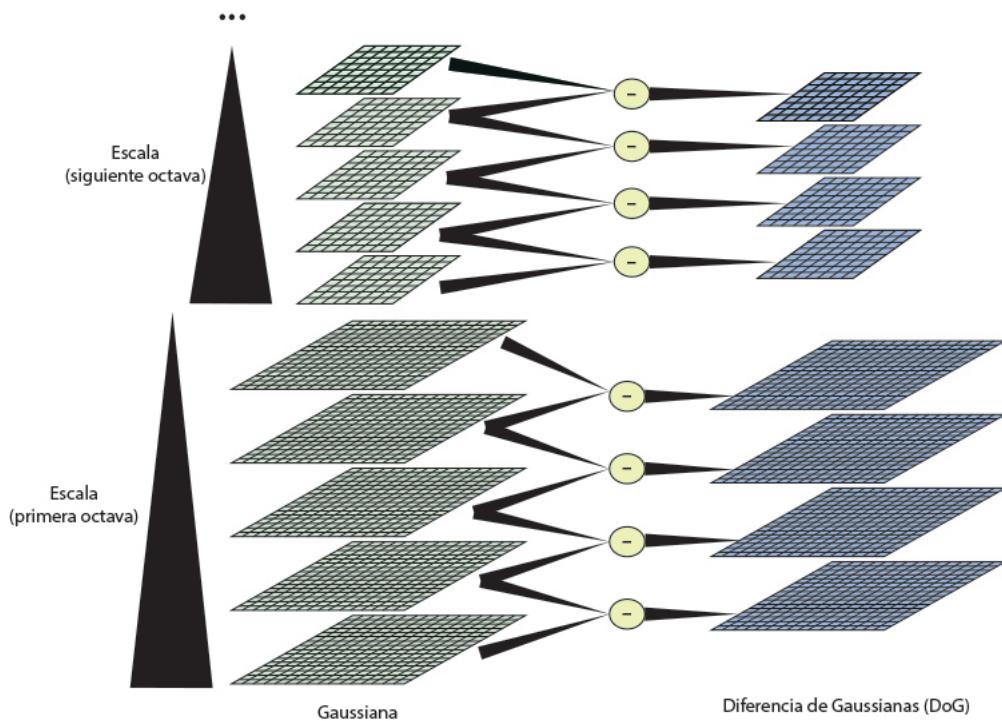


Figura 2-1: Espacio Escala de Diferencia de Gaussianas

podremos calcular por la diferencia de dos escalas cercanas separadas por un factor k multiplicativo:

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

$$= L(x, y; k\sigma) - L(x, y; \sigma)$$

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

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

Y por lo tanto,

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

En la figura 2-1 se puede ver la construcción de $D(x, y, \sigma)$. La imagen inicial se convoluciona con diferentes máscaras gaussianas, 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 entonces $k = 2^{\frac{1}{s}}$. Se producen $s + 3$ imágenes borronadas en la pila, por octava. Para extraer las

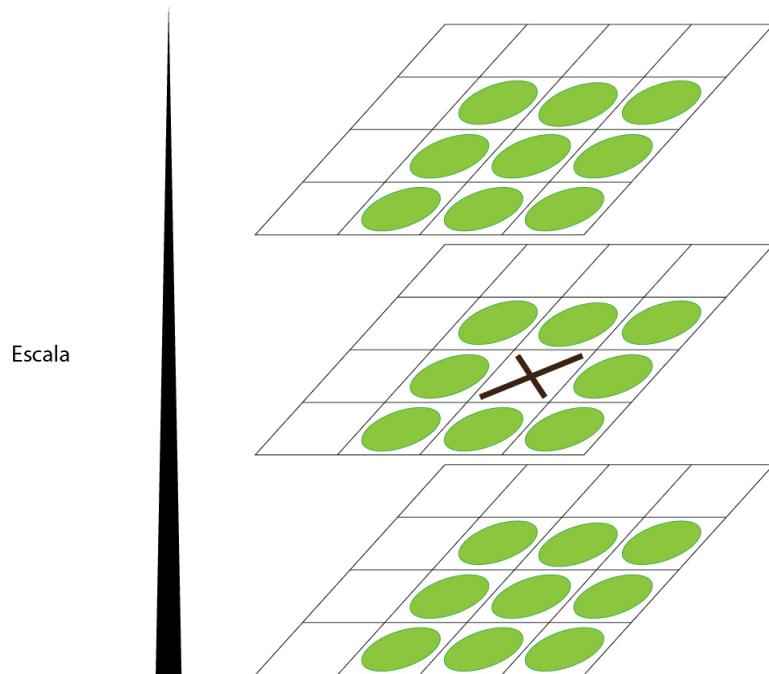


Figura 2-2: Espacio Escala de Diferencia de Gaussianas

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.

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 tendrían 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 $D(x, y, \sigma)$:

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

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 rechazara 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], encontraron experimentalmente que cualquier valor extremo menor de 0.03 es descartado:

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

Para el segundo caso, se utiliza una matriz Hessiana de 2×2 , H , 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 . 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 α el valor propio de mayor magnitud y β 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 α y β , $\alpha = r\beta$. Entonces:

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

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 cerciorar que la razón de las curvas principales es menor que cierto umbral, r , solo 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 encontremos 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 provocara 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 gaussianas:

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

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

Se formara un histograma de orientaciones que tendrá la orientación de los gradientes calculados en una región, al rededor 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.

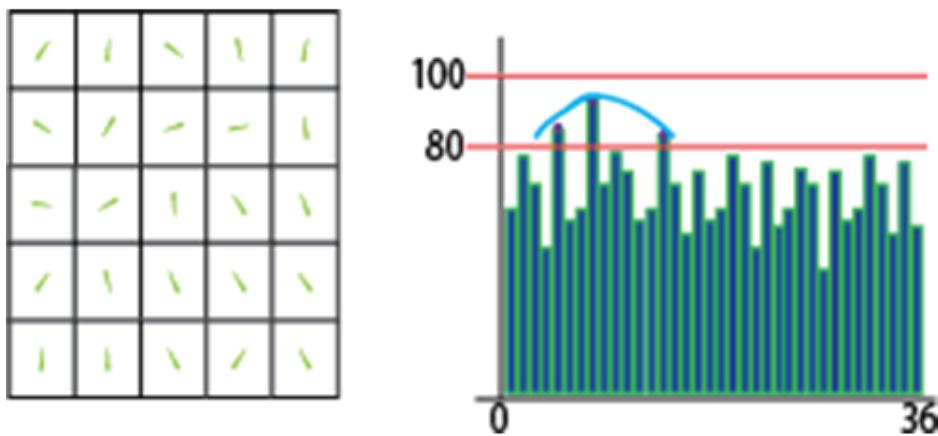


Figura 2-3: Histograma de Orientación

Para cada muestra agregada se ponderada por la magnitud de su gradiente y por una máscara circular gaussiana ponderada con σ , que es 1.5 veces que de la ubicación del espacio escala donde

2.1. EXTRACCIÓN Y DESCRIPCIÓN DE CARACTERÍSTICAS

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 tenga una orientación. 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.

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 16×16 pixeles y se dividirá en una región de 4×4 . Se generará un histograma de orientación de los gradientes de cada región, a diferencia del histograma de orientación explicado anteriormente, el histograma solo tiene 8 divisiones con las cuales se cubrirán los 360 grados, igualmente se usara una ponderación gaussiana para la asignación de la magnitud al histograma.

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

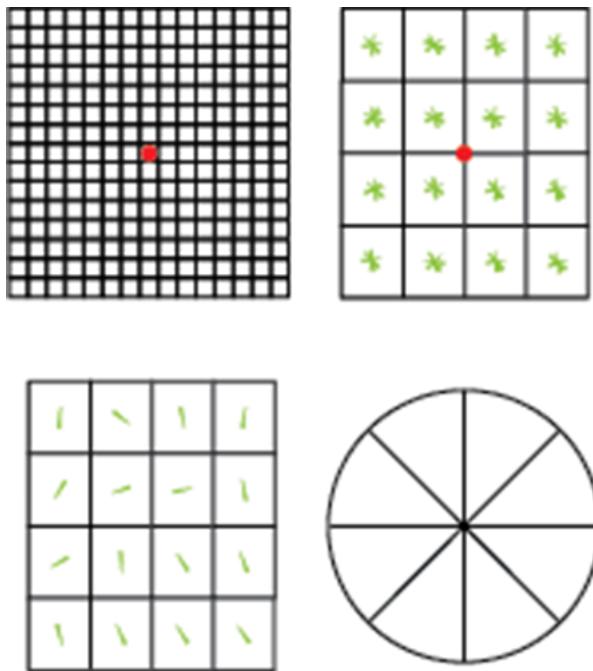


Figura 2-4: Descriptor

2.2. Computo en GPU

Hasta hace 12 años la velocidad a la que crecía cada generación de procesadores era increíble, los programas eran tan rápidos como cada nueva generación de procesadores. Este crecimiento entre cada generación se detuvo, el problema es el consumo de energía y la disipación de calor, no permiten aumentar la frecuencia del 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 de estas es comprensibles paso a paso, mediante el código. 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.

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.

Theoretical GFLOP/s

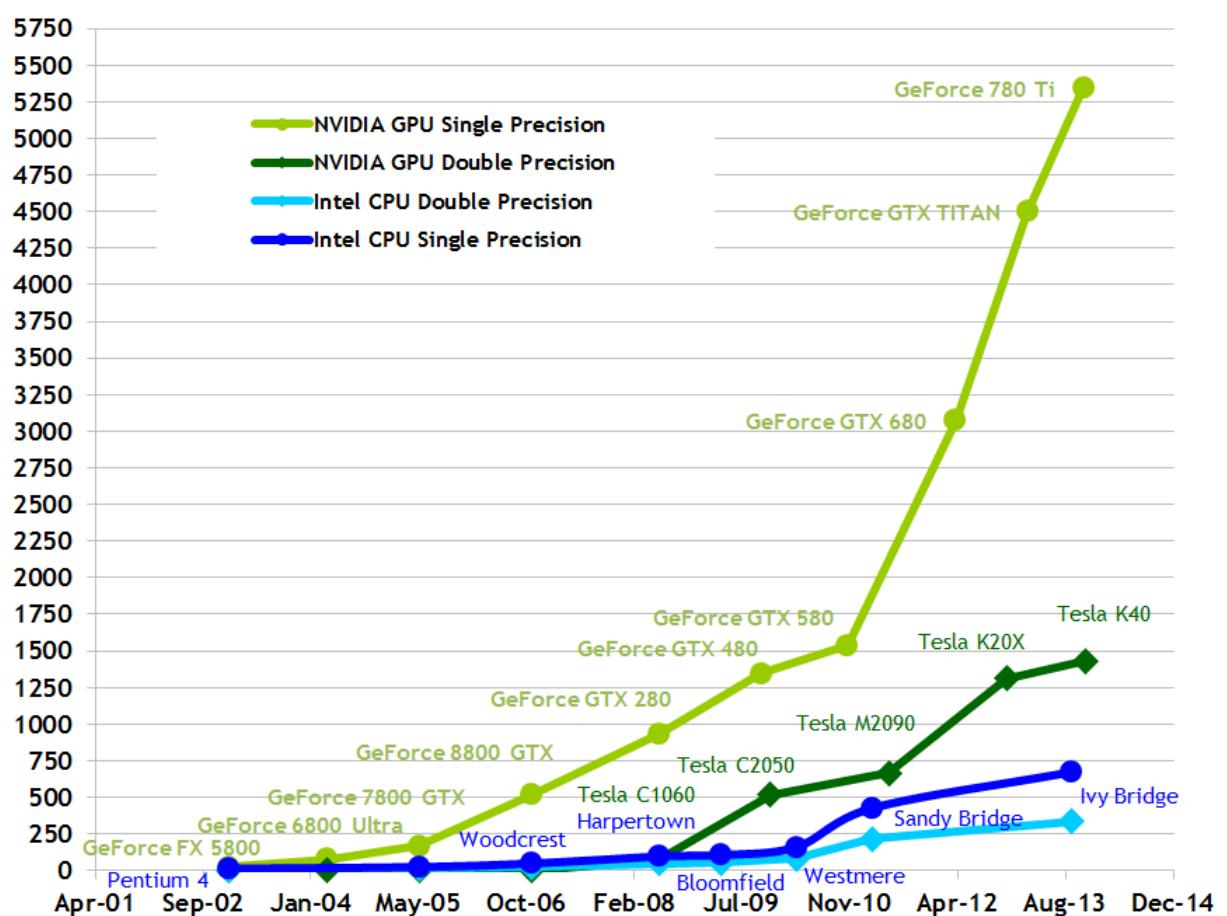


Figura 2-5: Comparación entre GPU vs CPU [3]

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 en paralelo, tiene 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 [4]. En la figura 2-5 podemos ver una comparación de estas dos corrientes, en cuanto a la cantidad de operaciones de punto flotante que pueden realizar en un segundo.

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

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”.[5]

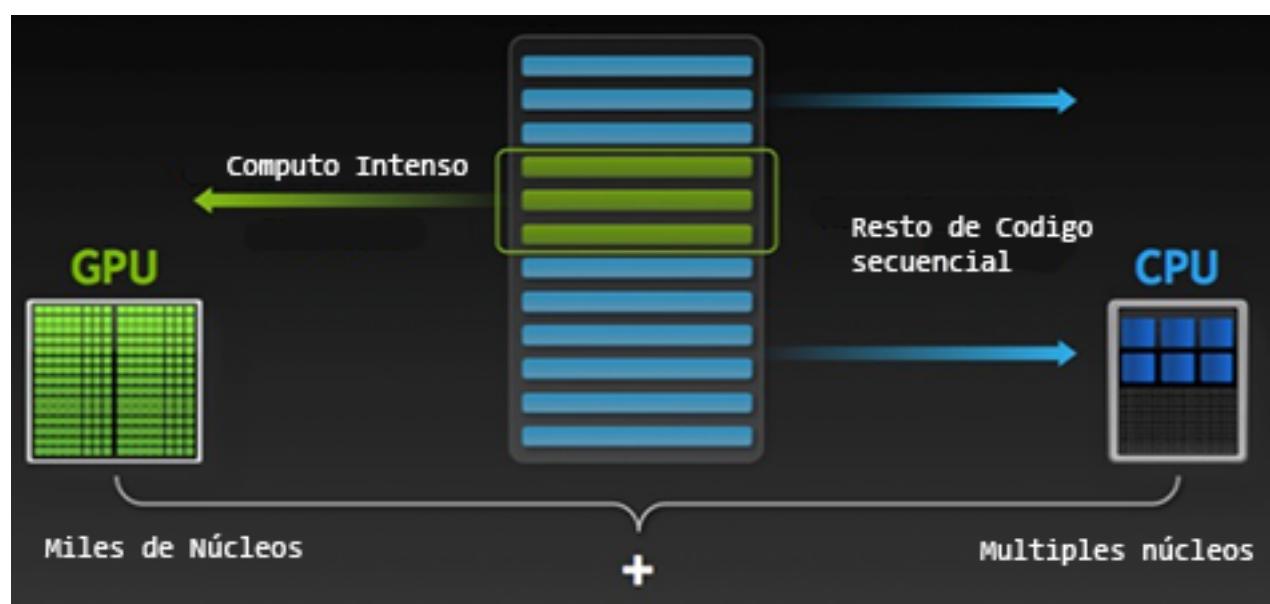


Figura 3-1: Sistema Híbrido

3.1. Breve Historia

La necesidad de mejores gráficos para los videojuegos, provocaron un gran avance en el hardware que se diseñaría. Desde principios de 1980 hasta finales de 1990 las tarjetas dedicadas a gráficos, no eran más que pipelines fijos que despliegan las formas geométricas calculadas por el CPU, por medio del hardware de acceso directo a memoria, por sus siglas en inglés DMA, esto les daba un funcionamiento fijo y apenas se podía configurar, con principalmente dos API, OpenGL de *Silicon Graphics* y Direct3D de *Microsoft*. Un ejemplo de estas tarjetas gráficas es, a la que se le acuño el nombre de GPU, la GeForce 256[6] lanzada al mercado en 1999, aporta una capacidad visual sin precedentes, capaz de realizar las funciones de transformación, iluminación, organización y rendering, con la capacidad de procesar 15 millones de triángulos por segundo y un rendimiento de 480 millones de píxeles por segundo. Su motor de rendering 256 bits muestra una mejora en cuanto a la complejidad visual.

Toda esta tecnología tan revolucionaria llamó la atención de otros profesionales, que se integraron a el trabajo de los artistas y desarrolladores de video juegos, utilizando el gran rendimiento de punto flotante que tenían los GPU para otros objetivos. De esta forma surge el movimiento de la GPU para fines generales(GP-GPU).

Pero en ese momento, la GP-GPU era muy difícil de manipular, solo aquellos que tenían amplios conocimientos en lenguajes de programación de gráficos, desarrollaban para esta plataforma. Pero aun que memorizara el API entera se enfrentaba un reto, donde los cálculos para resolver problemas generales debían ser representados por triángulos o polígonos.

Fue hasta 2001, en la Universidad de Stanford un equipo, liderado por Ian Buck, que se propuso ver el GPU como un *procesador de flujos*. Este equipo desarrollaría *Brook* [7], un lenguaje de programación diseñado para ser igual a la sintaxis de C, con algunas características adicionales. El lenguaje se desarrolla con el objetivo de minimizar el complejo trabajo de análisis, que se requería para generar aplicaciones paralelas. Introducirían conceptos como los flujos (streams), kernels y los operadores de reducción. Todo esto le dio un gran impulso a los GPU como procesadores de

propósitos generales, ya que el lenguaje era más fácil de manejar, ya que era de más alto nivel, y 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, pero debía complementarlo con herramientas de hardware y software intuitivas, con ello le hicieron la invitación a Ian Buck para colaborar con ellos, el objetivo sería ejecutar C a la perfección en una GPU. NVIDIA alcanza este objetivo en 2006 con el lanzamiento de CUDA, la cual sería la primera solución para las GP-GPU, y aunado a esta solución, lanza la GeForce 8800 la cual fue diseñada, para ser usada en cómputo de propósito general, y su arquitectura fue pensada en la de CUDA.

3.2. Plataforma y modelo de programación de computo paralelo

Compute Unified Device Architecture (CUDA) es una plataforma para computo paralelo y un modelo de programación, que NVIDIA lanzo en noviembre de 2006, permitiendo obtener aumentos en los rendimientos del cómputo, esto es 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 tienen una gran cantidad de datos a procesar, ya que la arquitectura de esta plataforma, de la cual se hablara adelante, es como un procesador tradicional como el que las computadoras tienen, solo que tienen la cualidad de que los procesadores son masivamente paralelos equipados con una gran cantidad de unidades aritméticas. En las cuales se ejecutara la misma instrucción en todas, respecto a la taxonomía de Flynn, la categoría seria de *una instrucción, múltiples datos* (SIMD).

Respecto al modelo de programación para desarrollar los programas para las GPU, es gracias a una extensión del lenguaje C, conocida como CUDA C. Existen alternativas a esta extensión, se pueden utilizar lenguajes como FORTRAN, Python, .NET combinando CUDA con Microsoft's F# o alguna API como OpenCL u OpenACC[8].

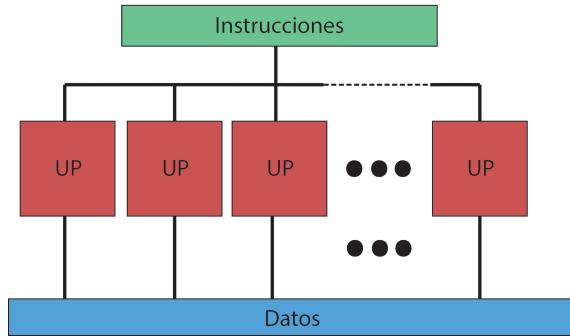


Figura 3-2: SIMD

3.2.1. Arquitecturas

La arquitectura de CUDA fue diseñada, para que la GPU pudiera ser utilizada en aplicaciones de propósito general. En la cual se tiene un arreglo de procesadores con múltiples unidades aritmético lógica, por sus siglas en inglés ALU, las cuales para alcanzar este objetivo, fueron diseñadas para poder realizar operaciones de punto flotante, cumpliendo los requisitos del Instituto de Ingeniería Eléctrica y Electrónica (IEEE). Aparte de esto las ALU debían 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 las conoceremos como *CUDA cores*, conforman gran parte de los Streaming Multiprocessor (SM). Los SM son procesadores que tienen la tarea de ejecutar los hilos concurrentemente, aparte de los CUDA cores tienen, están formados por una memoria cache (shared memory), registros y algunas unidades de funciones especiales.

Fermi

Los GPU basados en la arquitectura Fermi [9], están formados por 512 CUDA cores. Los CUDA cores ejecutan operaciones de punto flotantes o enteras por ciclo de reloj, y por cada uno de los hilos. Los 512 CUDA cores están organizados en 16 SM de 32 cores cada uno. El GPU tiene seis particiones de memoria de 64-bits, capacidad para leer 384-bits de la memoria simultáneamente y con 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 programación del trabajo

a realizar en cada bloque es asignado por un módulo llamado *GigaThread*, este pasa las tareas a cada SM para que el haga la asignación de trabajo a cada hilo.

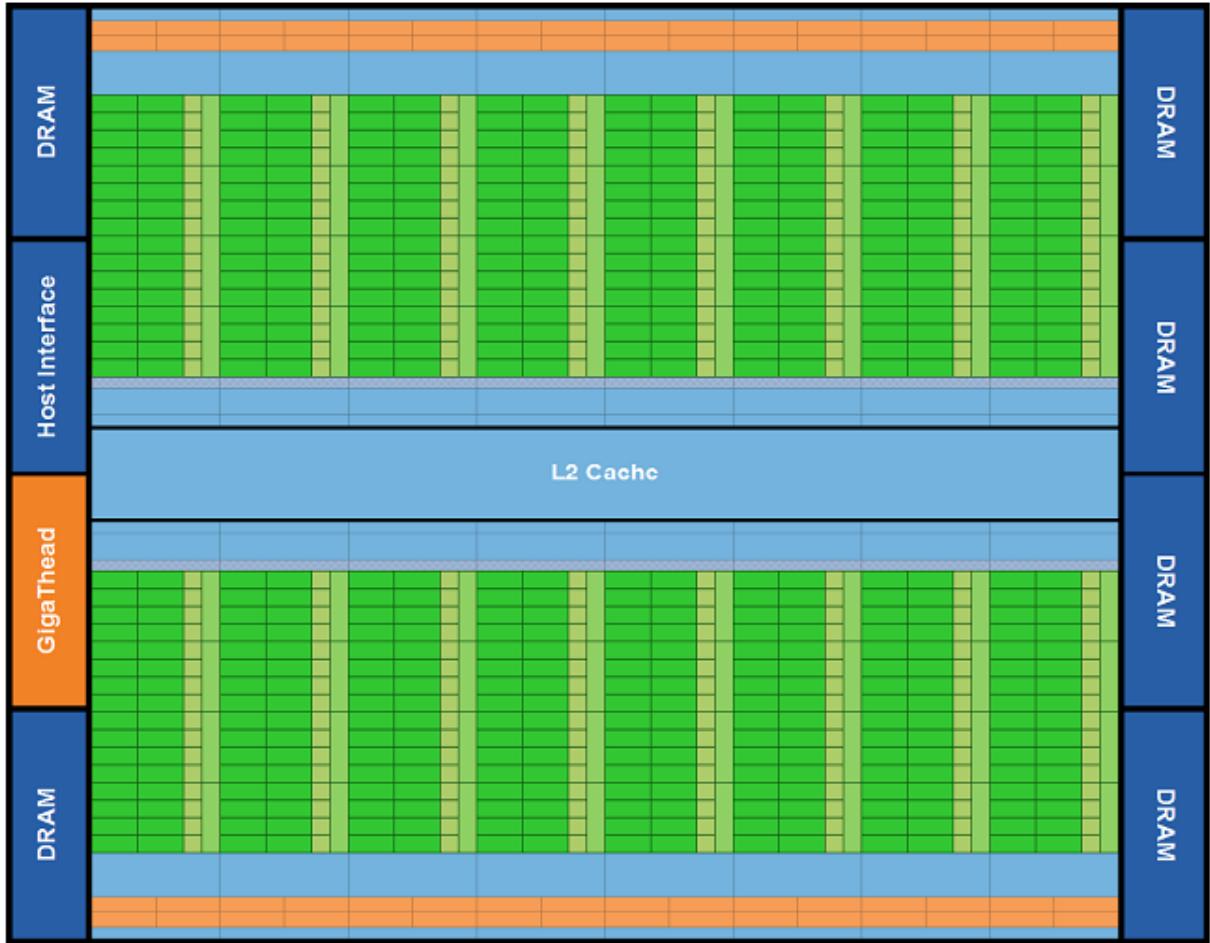


Figura 3-3: La arquitectura Fermi tiene sus 16 SM alrededor de la memoria compartida L2 cache [9]

La arquitectura tuvo mejoras significativas como el rendimiento en las operaciones de doble precisión, dedicado a computo científico; soporte para la corrección de errores, para asegurar las operaciones con números muy grandes, en aplicaciones delicadas; se implementó una jerarquía en la memoria cache, que permite 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 L2 cache.

Las SM de la arquitectura Fermi están formadas de diferentes elementos, iniciando 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

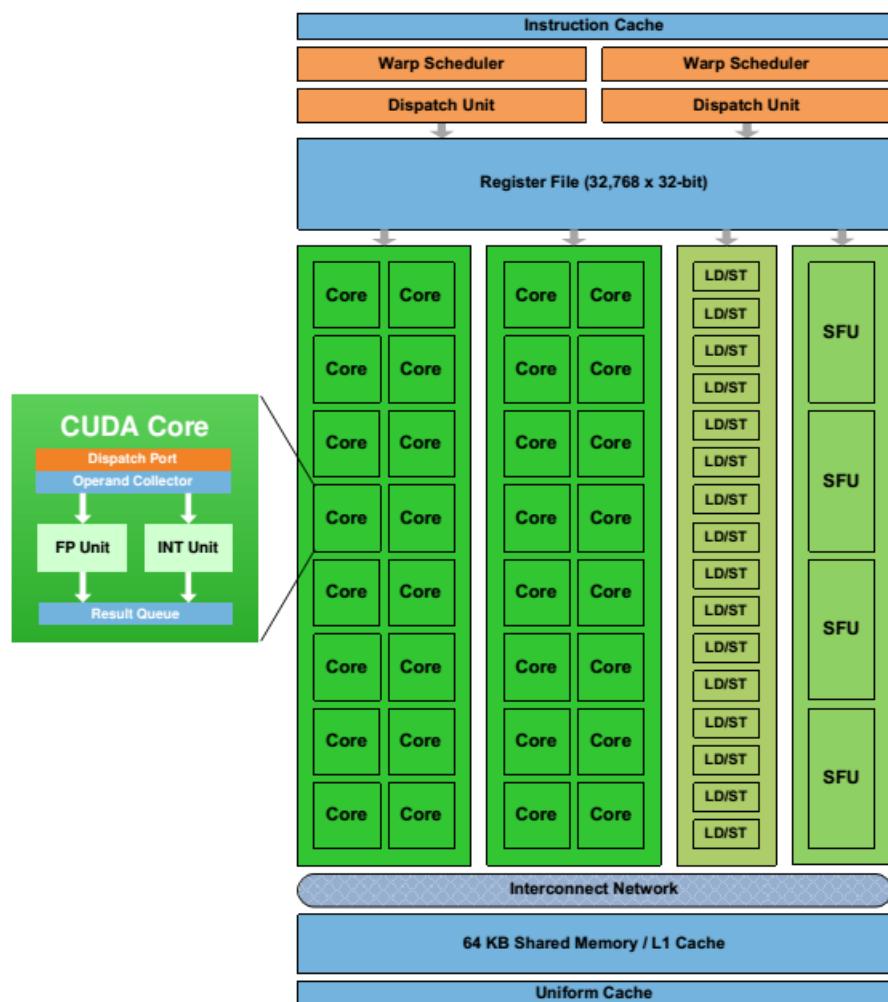


Figura 3-4: Fermi Streaming Multiprocessor (SM)[9]

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* es quien tienen el trabajo de dividir este bloque en grupos de 32 hilos para su ejecución dentro de la SM. También tiene 16 unidades load/store, las cuales permiten calcular origen y destino de dieciséis hilos por pulso de reloj; cuenta también con 4 unidades de funciones especiales (SFU), que ejecutan instrucciones más complejas como senos, cosenos, reciproco y raíz cuadrada.

Kepler

La arquitectura Kepler [10], modifico los SM de su antecesor llamándolo Next Generation Streaming Multiprocessor (SMX), es el nuevo procesador de esta arquitectura, en la cual encontraremos que está formada por 15 de estos procesadores y seis 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.

Las unidades load/store aumentaron a 32 y las SFU también incrementaron a 32, ocho veces más que en Fermi. La asignación de hilos dentro del SMX es programado igualmente por planeadores de warps, bloques de 32 hilos, Kepler tiene 4 planificadores de warps, de esta manera se tienen 2 unidades de despacho de instrucciones, permitiendo repartir y ejecutar 4 warps de manera concurrente. Nos encontramos con una memoria cache L1, la cual podemos cambiar su configuración.

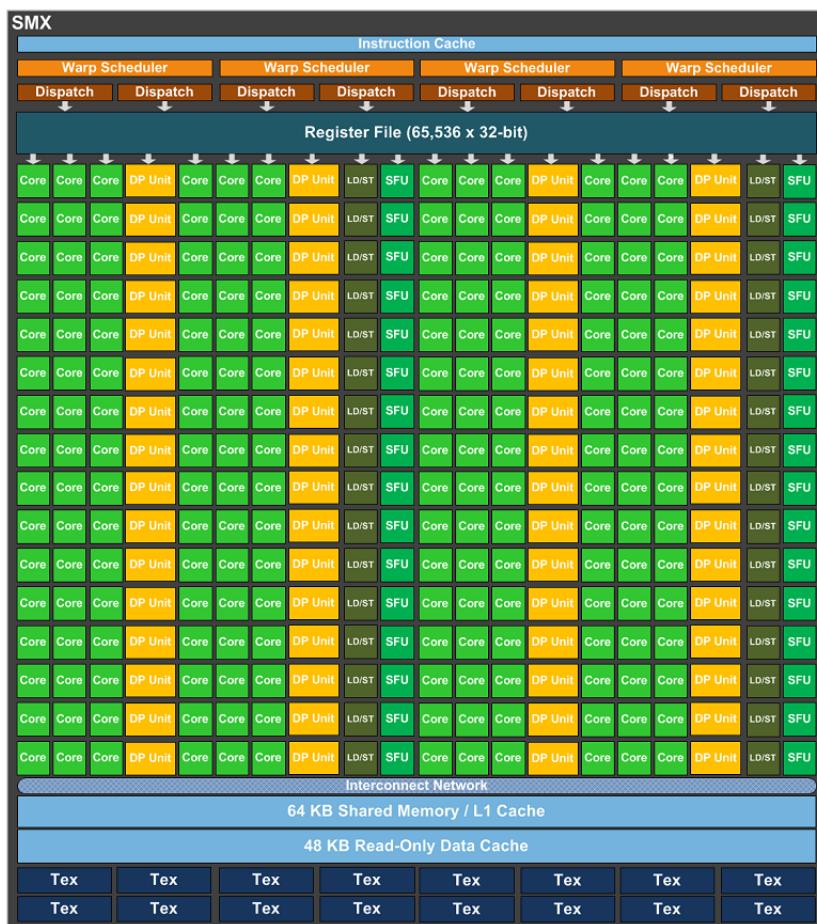


Figura 3-5: Kepler Next Generation Streaming Multiprocessor (SMX) [10]

La capacidad de esta memoria es de 64KB, se pueden tener las configuraciones de 16, 32 o 48 KB

para la memoria cache, dejando el resto para la memoria compartida. La cantidad de registros por SMX es de 65536, de los cuales, cada hilo puede tener acceso a 255 registros para el almacenamiento de datos. La memoria de textura ha sido un recurso valioso para programas donde se requiere probar o filtrar datos de una imagen, en esta arquitectura dejó de ser un hardware dedicado solo a este objetivo, se dejó un espacio en la memoria global de solo lectura de 48KB que funciona como una memoria cache para agilizar las lecturas. En esta arquitectura se agregó una característica, donde no se requiere del CPU para lanzar programas en la GPU, lo que significa que el 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 computo.

Maxwell

La arquitectura Maxwell[11], tuvo un cambio en su diseño para proporcionar un cambio dramático en su desempeño. Lo que generó este desempeño fue el nuevo diseño que le dieron a los nuevos SM llamados SM Maxwell (SMM). El número de CUDA cores bajó a 128, para poder separarlos en 4 divisiones de 32 CUDA cores, cada una de esas divisiones tiene un planificador de warps, para su bloque de 32 CUDA cores, el cual es capaz de despachar dos instrucciones por ciclo de reloj. Estas divisiones hicieron 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 incrementó a 96KB, la cual ya no se comparte con la memoria cache L1, ahora la memoria de textura comparte espacio con la cache L1. Los registros, las SFU, y las unidades load/store siguen siendo la misma cantidad.

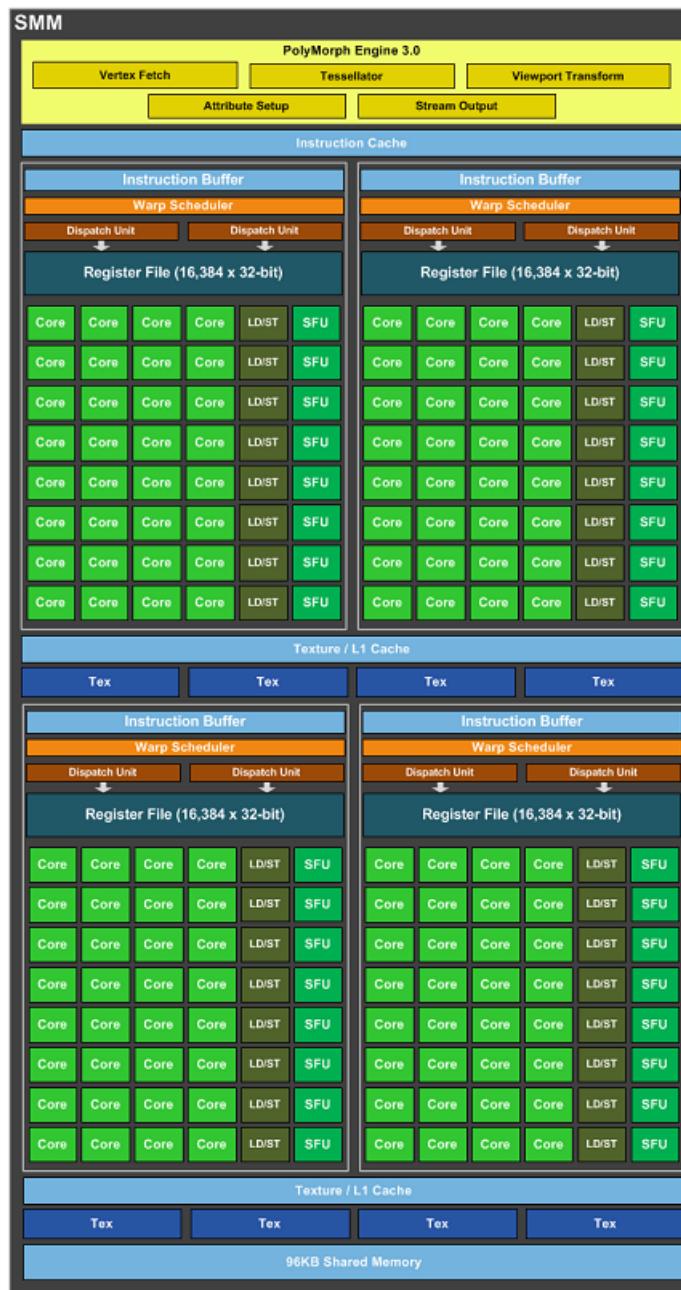


Figura 3-6: Maxwell Streaming Multiprocessor (SMM) [11]

3.3. Modelo de Programación CUDA C

La extensión del lenguaje C que proporciona CUDA para programar, es un camino que ofrece, para programadores familiarizados a este lenguaje, una manera sencilla de escribir programas para ser ejecutados en la GPU. A continuación se explicara el núcleo del conjunto de instrucciones de esta extensión.

3.3.1. Kernels

En CUDA C, permite definir funciones llamadas *kernels*, las cuales cuando son llamadas se ejecutan N veces en paralelo por 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 invocaran por medio de un equipo anfitrión, este anfitrión no es más que el procesador que estará usando la tarjeta gráfica como coprocesador. En el siguiente código podemos ver como se declara un kernel:

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

Listing 3.1: Declaración de un Kernel en CUDA C.

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

3.3.2. Jerarquía de Hilos

Para especificar la configuración que se usara para lanzar los hilos, se especifica poniéndolo entre `<<< y >>>`.Se requiere de dos parámetros para el lanzamiento el primero es la dimension de la malla, que se refiere al numero de bloques y el segundo es la dimension del bloque, que es el

numero de hilos que contendrá.

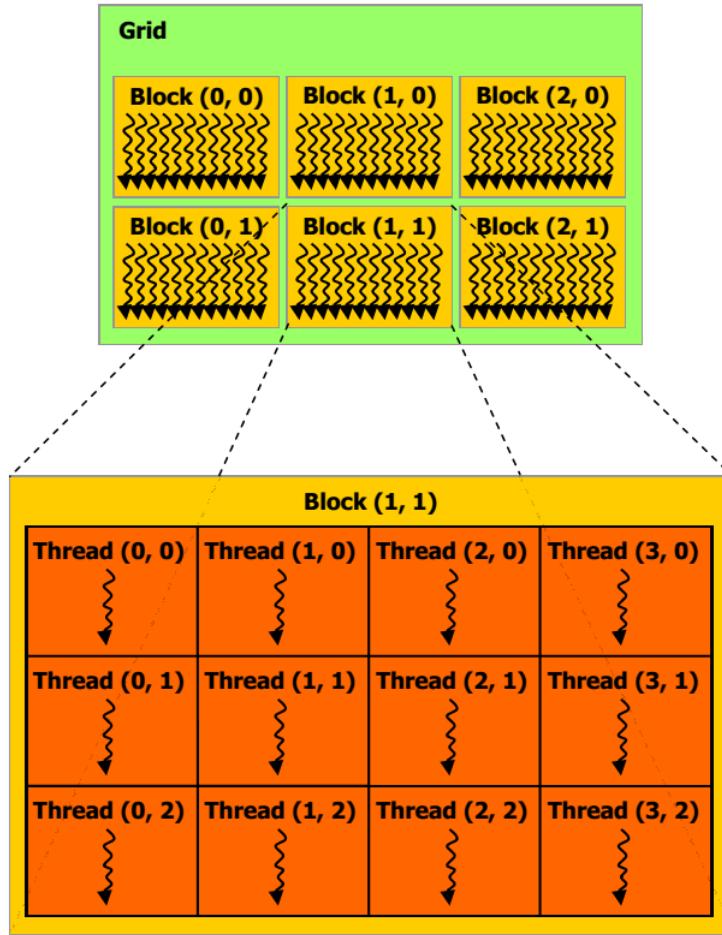


Figura 3-7: Organización de bloques e hilos [3]

Cada hilo tiene un identificador que se puede acceder por **threadIdx**, este identificador es un vector de tres componentes, por lo tanto los hilos pueden usar identificadores de uno, dos o tres dimensiones. Para formar bloques unidimensionales, dimensionales o tridimensionales. Los bloques están organizados en una malla que puede tener una, dos o tres dimensiones, los bloques tienen un identificador al cual se puede acceder por la variable **blockIdx**, existe otra variable importante y es la que nos dará la dimensión del bloque, se llama **blockDim**.

A continuación veremos un ejemplo de cómo se lanzaría un kernel:

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

Listing 3.2: Lanzamiento de un Kernel en CUDA C.

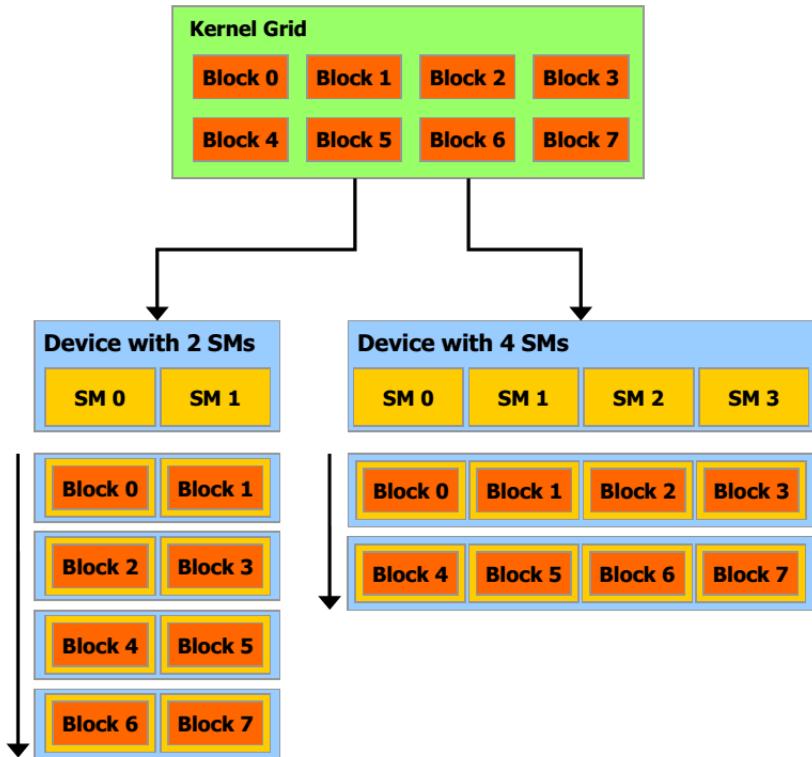


Figura 3-8: Asignación de bloques por SM [3]

Parte importante del paralelismo es la comunicación entre cada proceso que se ejecuta simultáneamente, hacer cooperar los hilos en la GPU tiene sus detalles. Para la comunicación tenemos la memoria compartida a la cual permite el intercambio de datos solo entre hilos del mismo bloque. En cuanto a sincronizar los hilos se tiene una función llamada `_syncthreads()`, esta sincroniza los hilos por barrera, pero los hilos que esperan solo lo harán con hilos de su mismo bloque. Esta característica de que solo hilos del mismo bloque se puedan comunicar es por la forma en que son asignados para su ejecución cada uno de los bloques. En la figura 3-8 podemos observar cómo se asignan los bloques a cada SM por lo tanto podrían asignarse en cualquier orden y ejecutarse en tiempos diferentes. De este modo la sincronizar los hilos y la escritura y lectura a memoria compartida no serán un problema con el diseño que se describió anteriormente.

3.3.3. Jerarquía de Memoria

Los hilos de un kernel son los encargados de realizar las operaciones sobre los datos, se tiene diferentes tipos de almacenamientos en la arquitectura CUDA de los cuales se pueden leer los

3.3. MODELO DE PROGRAMACIÓN CUDA C

datos para operar y escribir los resultados obtenidos. Cada uno de estos almacenamientos tiene características diferentes, que nos ayudaran a mejorar el despeño de los programas. A continuación hablaremos de los tipos de memoria que se encuentran en las GPU de Nvidia.

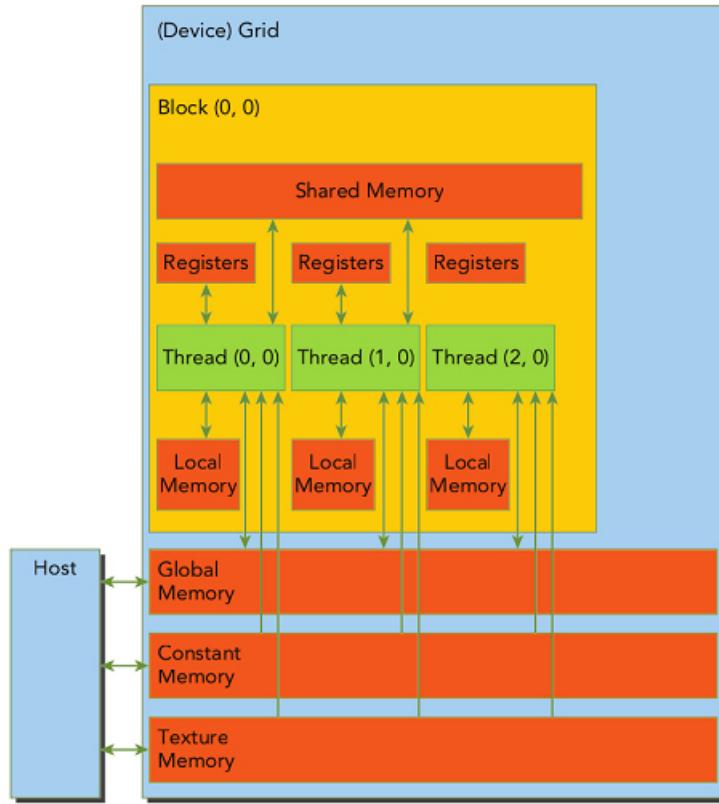


Figura 3-9: Tipos de Memoria [12]

Los *registros* son el tipo de memoria con la lectura/escritura más rápida que ninguna otra en el dispositivo. En cada SM tenemos miles de registros y a cada hilo se les asigna una cantidad de estos registros, cuando es lanzado el kernel. Los registros son de 32-bits en los cuales se pueden almacenar datos de tipo flotante o entero. La manipulación de este espacio esta administrado por el sistema.

La *memoria local* es un espacio de memoria privada que cada hilo tiene, podemos ver que se almacenan datos que no pudieron ser almacenados en los registros como variables locales, 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 cache que se comparte entre hilos de un mismo

3.3. MODELO DE PROGRAMACIÓN CUDA C

bloque, esto genera que los hilos del bloque puedan comunicarse, escribiendo y leyendo en ella, para cooperar en la realización de un mismo objetivo. Este cache es especial ya que el programador elige su manejo, la forma en la que se declara una variable en este espacio, es con ayuda de la palabra reservada **_shared_**. La latencia en esta memoria es hasta 100 veces menor en comparación con la memoria global.

La *memoria constante* es una memoria de solo lectura, en la cual albergaremos datos que no cambiaran a lo largo de la ejecución del kernel. Podemos ubicar esta memoria en el dispositivo, al igual que la memoria global, pero esta 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. Con la palabra reservada **_constant_**, se puede declarar la variable, pero el contenido de este debe ser asignado por el anfitrión, en la memoria del dispositivo, antes de lanzar el kernel, con ayuda de 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 un lecturas eficientes en arreglos de una, dos o tres dimensiones. Las lecturas en este tipo de memoria tienen ventajas como diferentes formas de acceso e interpolaciones en los datos que se pueden utilizar sin costos adicionales.

La *memoria global* es la memoria de lectura/escritura de mayor tamaño en la tarjeta gráfica, llega al orden de los gigabytes. Las funciones que tiene son de 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 será, hasta que se liberen, lo que nos permite que esta memoria funcione para compartir datos entre kernels. Los hilos pueden acceder en cualquier momento del kernel a esta memoria, pero su latencia es tan alta que podría provocar que tarde más la lectura de datos que los cálculos que se quieren realizar. Existen funciones para reservar, manejar y liberar el espacio en memoria, desde el anfitrión: **cudaMalloc()**, **cudaMemCpy()** y **cudaFree()**.

3.3.4. Programación heterogénea

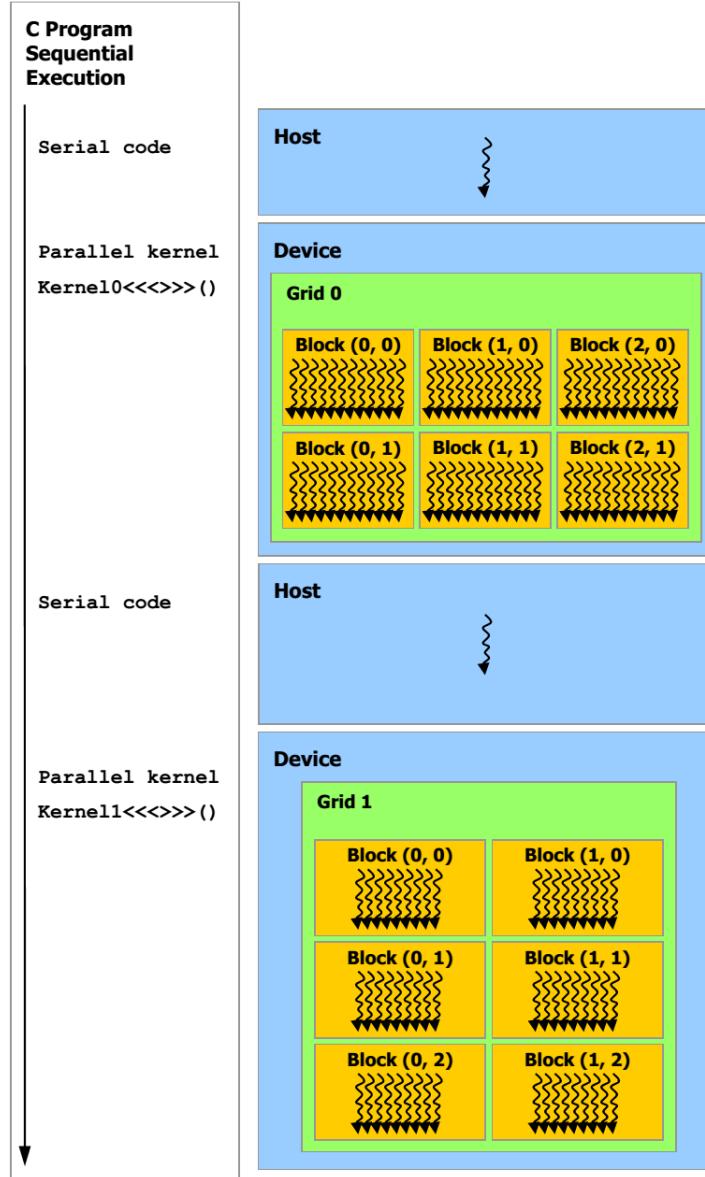


Figura 3-10: Programación Heterogénea [3]

Para entender el modelo de programación de CUDA, se tiene que tener en cuenta que se debe hacer código para el CPU y el GPU para que estos puedan trabajar en conjunto. El CPU es el equipo anfitrión (Host) el que decidirá cuándo necesitará usar al dispositivo (Device) GPU, el anfitrión y el dispositivo también tendrán memorias separadas.

Un Programa en CUDA C se ejecuta como se muestra en la figura 3-10, en el siguiente código se puede ver la estructura básica de un programa, en el cual podemos ver que se declara y definen funciones kernel, también podemos ver que hay funciones ejecutables en el GPU, siendo llamadas

3.3. MODELO DE PROGRAMACIÓN CUDA C

desde algún kernel se definirán con la palabra reservada `__device__`.

En la función principal el anfitrión se encargara de obtener los datos que se le proporcionaran y almacenarlos al kernel para ser procesados por el GPU, también podemos ver cómo es que se reservara la memoria en el dispositivo para los datos de entrada y salida que el kernel necesite para procesarlos.

Después de realizar la copia de los datos de anfitrión a dispositivo, se pueden ejecutar uno o más kernels en el GPU, una vez finalizada la ejecución de estos kernels el resultado se copia a la memoria del anfitrión.

3.3. MODELO DE PROGRAMACIÓN CUDA C

La final solo queda liberar los recursos que ya no serán utilizados.

```
1      __device__ L funcionDevice()
2
3      { ... }
4
5      ...
6
7      L r= fooDevice();
8
9      ...
10
11     __global__ void KernelUno(L*, ... )
12
13     { ... }
14
15     int main(...)
16
17     { ...
18
19     L* datosD;
20
21     cudaMalloc(&datosD, size);
22
23     ...
24
25     cudaMemset(datosD, 0, size);
26
27     ...
28
29     dim3 gridDim(..., ..., ...);
30
31     dim3 blockDim(..., ..., ...);
32
33     KernelUno<<<gridDim,blockDim>>>(datosD, ... );
34
35     ...
36
37     KernelDos<<<..., ... >>>(... );
38
39     ...
40
41     cudaMemcpy(res, datosD, size, cudaMemcpyHostToDevice);
42
43     ...
44
45     cudaFree(datosD);
46
47     ...
48
49     return 0;
50 }
```

```
26         cudaFree(datosD);  
27         ...  
28     }
```

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

CAPÍTULO 4

SIFT EN GPU

La correcta paralelización de un algoritmo no es nada trivial. Después del capítulo anterior, al ver todas las ventajas que tenemos en los GPU podríamos 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 podrían parallelizar. Para saber cómo analizar si un algoritmo es paralelizable primero se dará una definición de qué es un programa paralelo:

"Programa paralelo es la especificación de dos o más procesos simultáneos que cooperan entre sí con un fin en común"

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, es simplemente como organizar a los procesos para que mientras realizan su parte de trabajo sin que se interfieran entre ellos. Entonces se debe cambiar la forma en que se hacen los programas, ahora no solo es el como llegar a un objetivo paso a paso, sino que se tiene que pensar como muchos procesos trabajarán juntos para alcanzar un objetivo, esto inmediatamente da la idea de repartir o dividir el trabajo entre todos ellos.

Así que para realizar la labor de paralelizar el algoritmo se analizan básicamente tres casos de paralelismo:

- *Funcional*: Aquí 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.

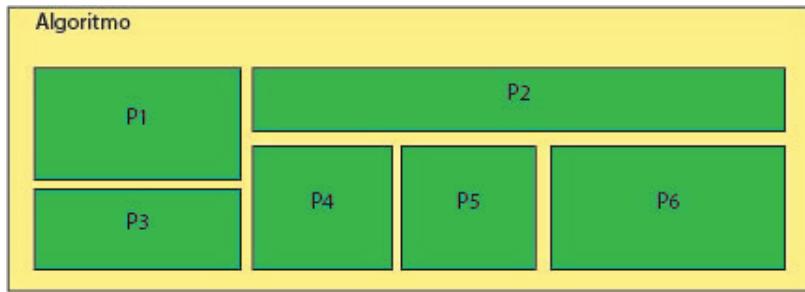


Figura 4-1: Todos los procesos son partes diferentes del algoritmo

- *Dominio*: Se repartirán los datos en los múltiples procesos los cuales tienen una especificación idéntica. El sincronizar es sencillo en este caso, pero aun así hay que presentar atención ya que podríamos corromper información.
- *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 analizarán las partes de SIFT para de esta manera adaptarlo al modelo de programación de CUDA.

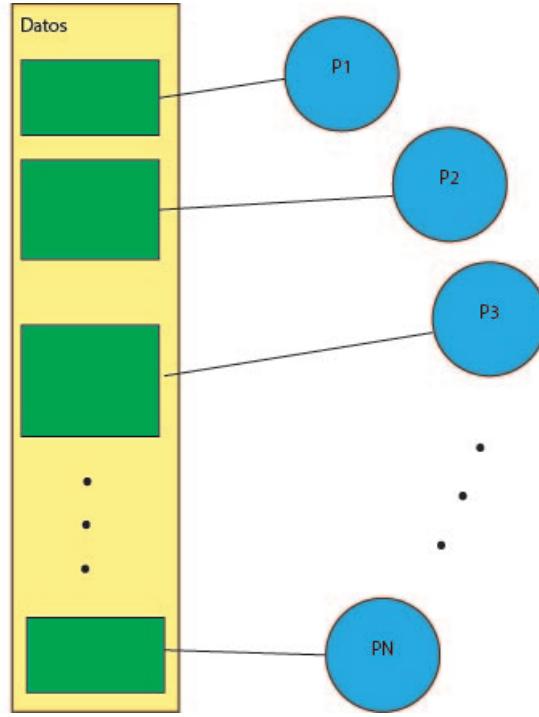


Figura 4-2: Todos los procesos tienen la misma especificación

4.1. Análisis de SIFT para su Paralelización en GPU

En esta sección se describe como se comunicaran y sincronizar los procesos, así como la estructura que tomara 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, estas partes no se ejecutarán simultáneamente, es solo que el algoritmo es bastante largo y al separarlo en estas partes se simplifica en diferentes kernels, que tendrán una secuencia de ejecución.

Los kernels de las diferentes partes del algoritmo tienen una estructura en común, básicamente todos tiene como entrada una o más imágenes, las cuales serán de solo lectura, y tendrán como salida una imagen o varias de salida. Cada proceso tendrá una sección de la imagen de tamaño $N \times N$, la cual puede estar traslapada con la de algún otro proceso, pero esta área no requiere de sincronizar entre procesos ya que solo será para obtener datos, para procesarlos. En la imagen de salida el proceso se le será asignado solo un pixel de la imagen para escribir, como se puede mostrar en la figura 4-4 las zonas del P1 y P2 están traslapadas y en la imagen de salida, que es como si tuviera un zoom a los pixeles, no escriben en otro que no sea su pixel.

4.1. ANÁLISIS DE SIFT PARA SU PARALELIZACIÓN EN GPU

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

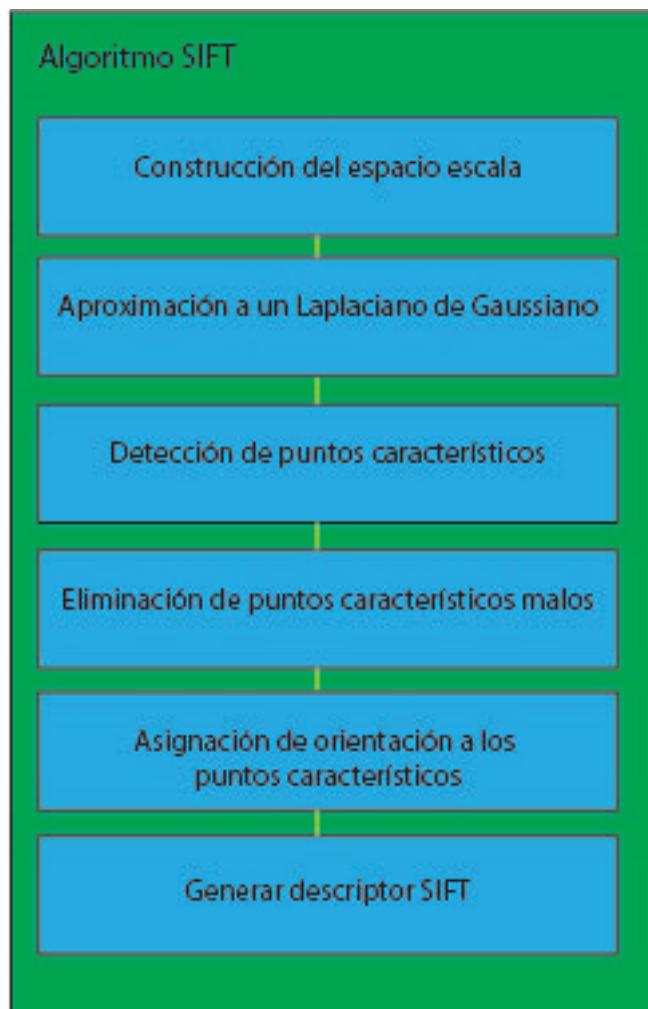


Figura 4-3: División del algoritmo SIFT a paralelizar

Cada uno de los kernels serán ejecutados múltiples veces, este trabajo será desempeñado por el anfitrión (CPU) de forma secuencial, esto es importante ya que cada uno de estos kernels es lanzado sin importar que el anterior acabara de ejecutarse, si múltiples kernels son lanzados y tiene 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 iguales 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

Teniendo en cuenta la estructura general de la solución, se plantea como construir cada una de las partes en las que dividí el algoritmo de SIFT, figura 4-3, para poder adaptarlo a el modelo de programación de CUDA.

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

Recordando un poco, el espacio escala se forma a partir de suavizar la imagen original a diferentes niveles de detalles, lo que se hizo fue cambiar el filtro gaussiano por uno que se aproxima a el Laplaciano de Gaussiana, que sería una diferencia de Gaussianas.

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

Para ello se obtienen los filtros Gaussianos con la ayuda de la librería OpenCV [13], y se restan para así obtener los filtros que aplicaremos en cada octava. Quedando como los de la figura 4-6.

También se uso OpenCV para cambiar el tamaño de las imágenes, para cada octava.

La sección , de las que se dividió el algoritmo, que es altamente paralelizable es al momento de hacer la convolución. Por este motivo lo que hice fue desarrollar un kernel, se puede encontrar en el apéndice A, que realice la convolución de una imagen y un filtro.

La idea es que estos kernels son lanzados por el anfitrión, se lanzaran tantos kernels como imágenes se necesitan para crear el espacio escala. Los kernels serán ejecutados de forma concurrente.

La funcionalidad del kernel es, para cada pixel en la imagen de salida se asigna un hilo de diferentes bloques, y estos se encargarán con los datos de entrada, la imagen y el filtro, implementar la operación de convolución.

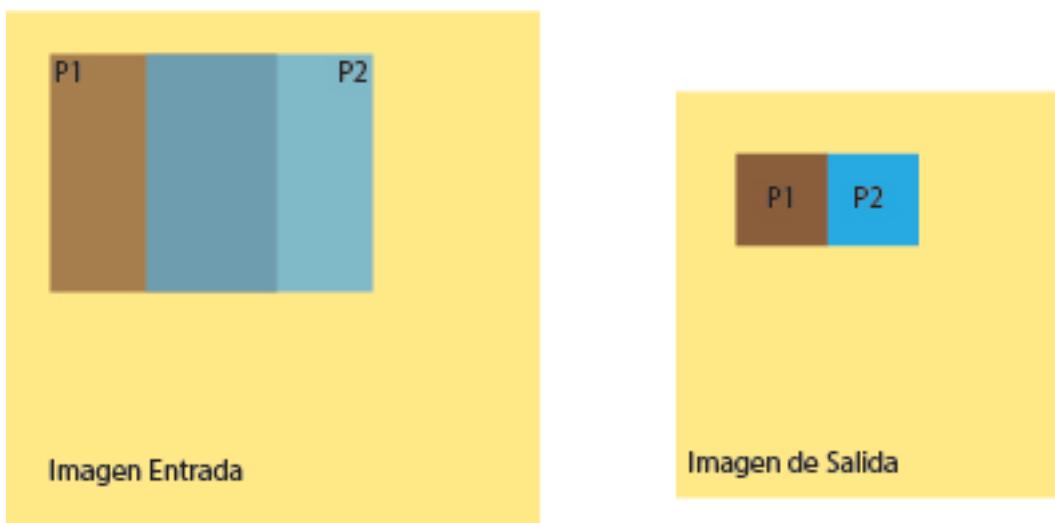


Figura 4-4: Proceso general de los kenels

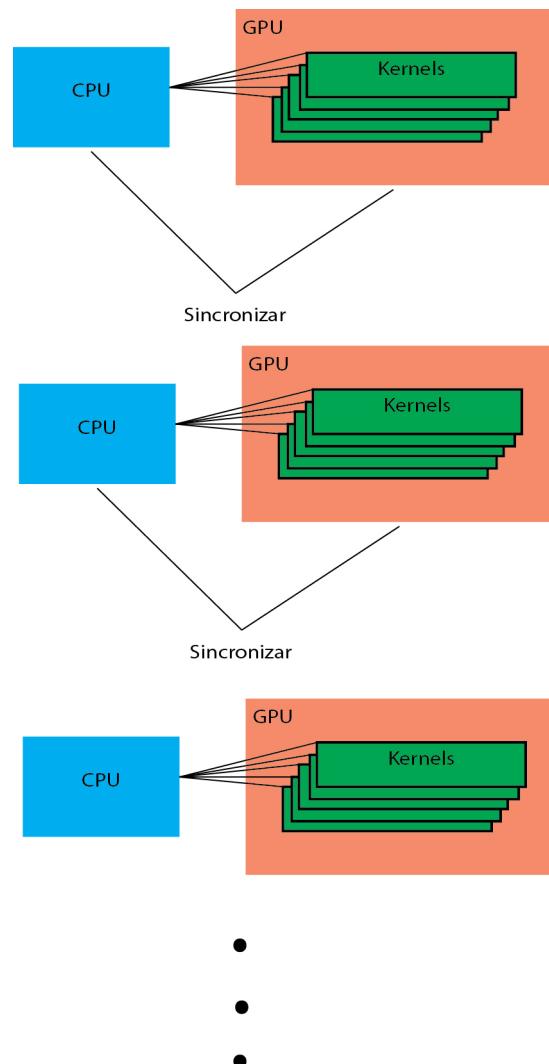


Figura 4-5: Lanzamiento de Kernels



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

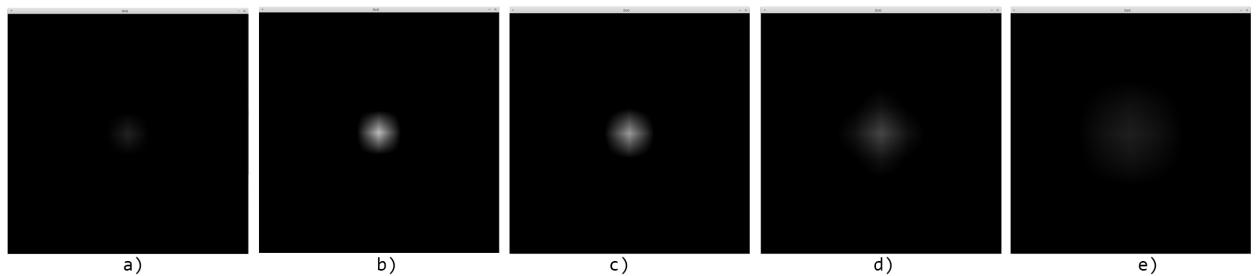


Figura 4-7: Filtros de diferencias de gaussianas. a) $\sigma_0 = 1 - \sigma_1 = 1.08$, b) $\sigma_1 = 1.08 - \sigma_2 = 1.36$, c) $\sigma_2 = 1.36 - \sigma_3 = 1.72$, d) $\sigma_3 = 1.72 - \sigma_4 = 2.16$, e) $\sigma_4 = 2.16 - \sigma_5 = 2.72$.

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: Calculo de la convolución para cada imagen del espacio escala
 El resultado es como el de la figura 4-7, tendremos imágenes muy similares para cada una de las octavas.



Figura 4-8: Espacio Escala de Diferencia de Gaussianas

4.2.2. Detección de puntos característicos

En el espacio escala, obtenido anteriormente, se buscan los puntos extremos. En $D(x, y, \sigma)$ se buscan 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. Solo 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

4.2. IMPLEMENTACIÓN

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

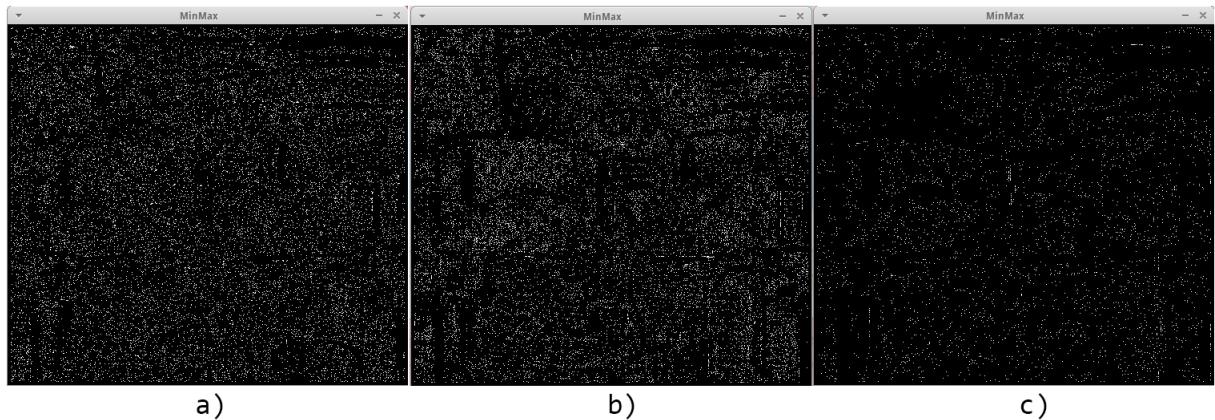


Figura 4-9: Mascara. Búsqueda con las imágenes del espacio escala: a) 0,1,2 ; b) 1,2,3; c)2,3,4

La imagen de salida terminara siendo una máscara binaria si existe un punto extremo, pondrá un pixel en blanco y donde no lo dejara en negro, como se muestra en la figura 4-8; Se puede encontrar la implementación de este algoritmo en el apéndice B.

4.2.3. Eliminación de puntos característicos malos

Ahora se filtraran los puntos extremos encontrados anteriormente. Existen dos casos donde los puntos extremos anteriormente seleccionados tendrían que ser eliminados:

1. El punto tiene un contraste muy bajo.
2. El punto está localizado sobre 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 4-9 pero esta vez se tienen muchos menos pixeles en color blanco. En el apéndice C podemos ver más detalladamente como es que se realizó la implementación de esta parte del algoritmo de SIFT.

4.2. IMPLEMENTACIÓN

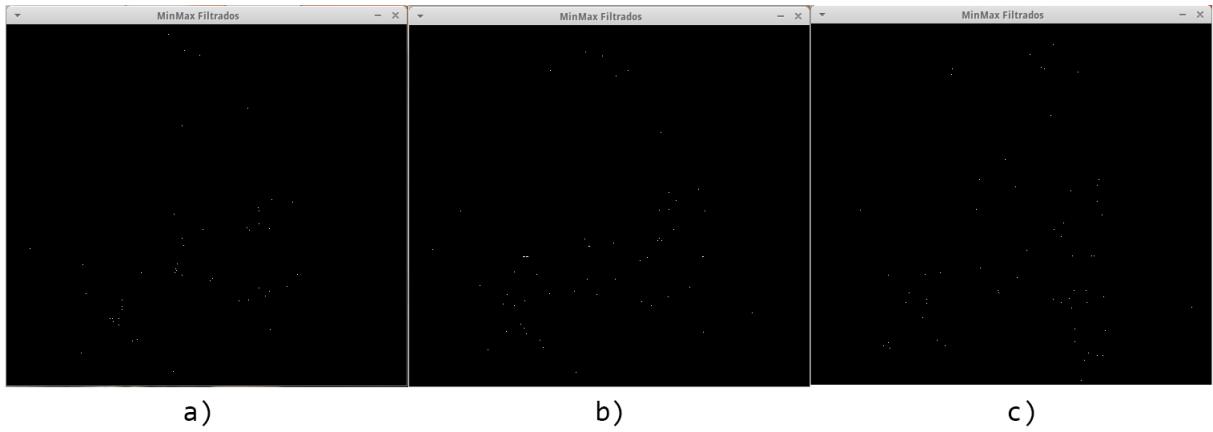


Figura 4-10: Mascara Filtrada de la búsqueda con las imágenes del espacio escala: a) 0,1,2 ; b) 1,2,3; c)2,3,4

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: Eliminación de puntos característicos malos

Para esta parte lo mejor fue dividir en dos kernels este proceso uno para calcular las magnitudes y orientaciones de los gradientes. En el apéndice D se encuentra la implementación de este kernel.

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

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

4.2. IMPLEMENTACIÓN

Y la otra será donde se obtienen los histogramas para cada punto característico y asignar una orientación dominante. En el apéndice E, se encuentra su implementación.

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: Eliminación de puntos característicos malos

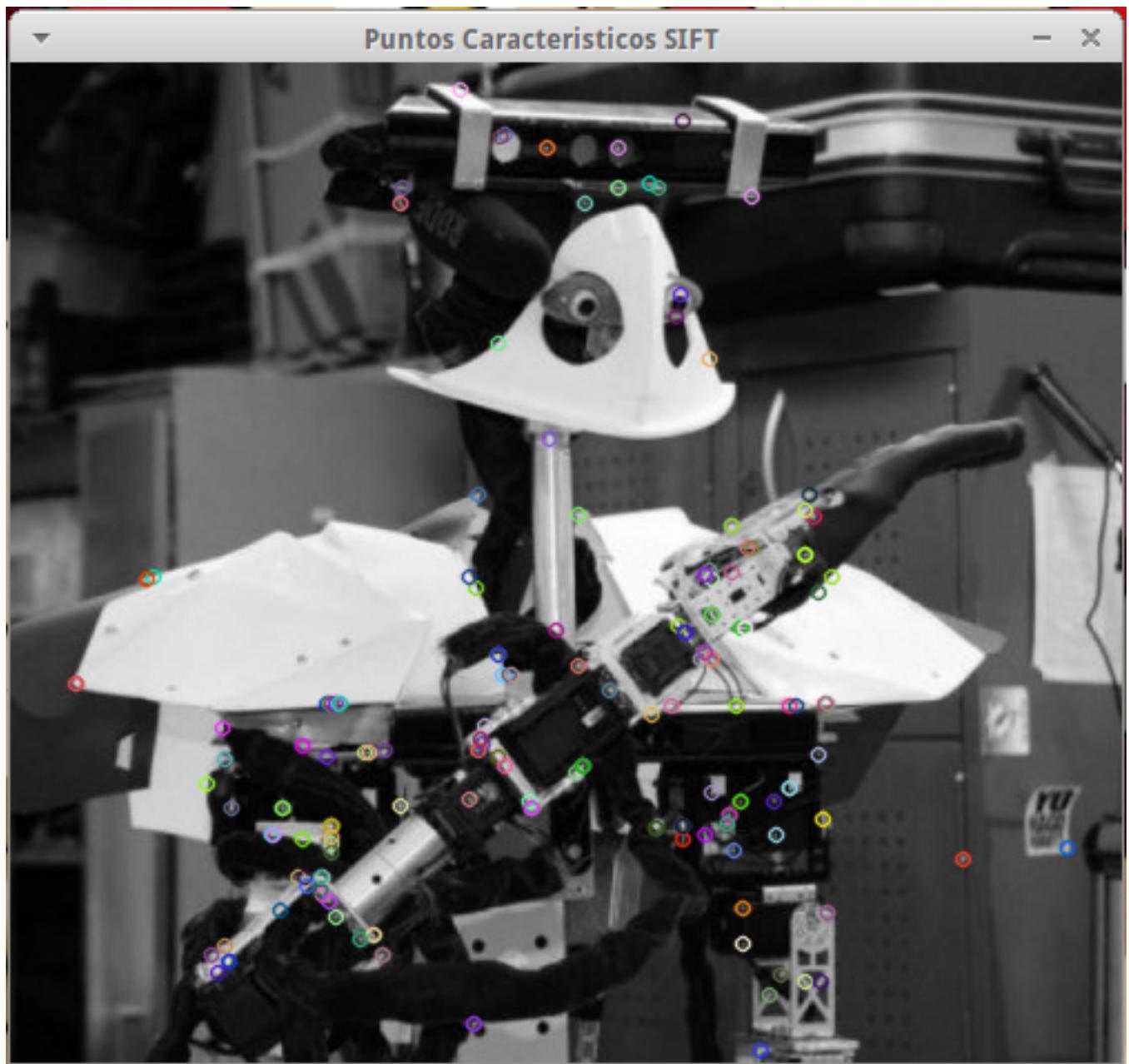


Figura 4-11: Puntos característicos

CAPÍTULO 5

PRUEBAS Y RESULTADOS

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 se hicieron bajo un sistema operativo xubuntu 14.04, utilizando opencv y CUDA 6.5.

Hay dos imágenes que fueron las que ayudaron a realizar paso a paso este desarrollo, la primera es la de la Figura 5-1 es un castor y la segunda es la de la Figura 5-2 de un gato. La razón de por la cual se tomaron estas dos imágenes fue por que la del castor es una imagen pequeña y la del gato es bastante grande, a su vez el castor cuenta con pocos puntos característicos contrario al gato que en todas las pruebas que realice fue el que más puntos característicos tiene. Lo que se da a entender es que se tomaron los extremos en cuanto a los datos de entrada.

La primer prueba que se hizo con el castor y el gato, consistieron en medir el tiempo que tardaban en ejecutarse ciertas secciones del código de una implementación abierta de SIFT llamada Open SIFT [14], la cual se usa para la detección de objetos en el robot Justina, y en la implementación de SIFT con CUDA que realice para obtener los puntos característicos. Se puede observar en las tablas 5-1 y 5-2 los resultados de estas pruebas. Las unidades del tiempo son en milisegundos.

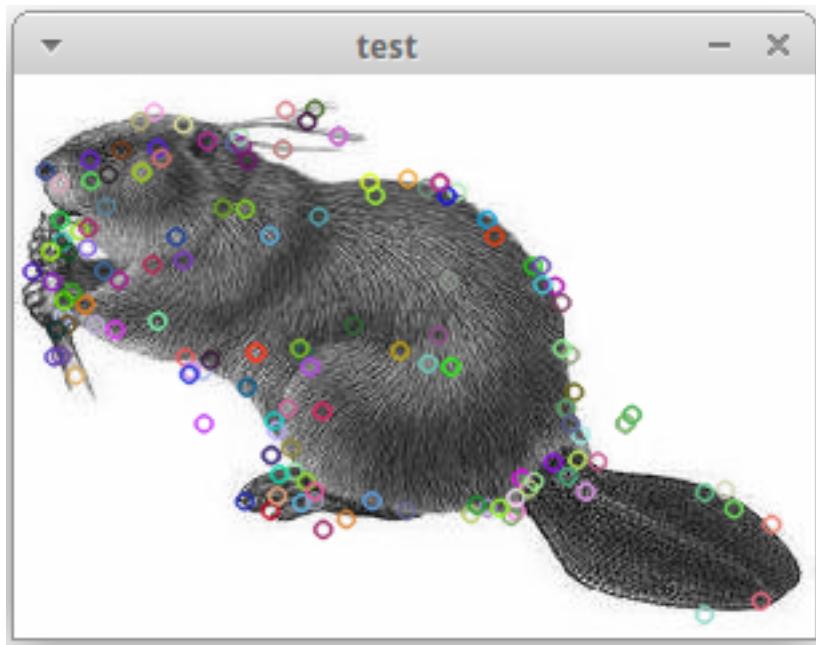


Figura 5-1: Puntos característicos encontrados en la imagen del castor



Figura 5-2: Puntos característicos encontrados en la imagen del gato

Castor		
Partes de SIFT	CUDA SIFT	Open SIFT
Espacio escala DoG	17.25	29.32
Detección de PC	1.87	50.09
Eliminación de PC malos	0.75	
Orientación de PC	8.80	12.35

Tabla 5-1: 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	957.19
Detección de PC	55.47	2210.82
Eliminación de PC malos	9.82	
Orientación de PC	125.2	1014.89

Tabla 5-2: La resolución de la imagen es de 1920x1200 px y se encontraron 12000 puntos característicos

Lo que se puede notar de estos resultados de las tablas 5-1 y 5-2 es la aceleración al crear el espacio escala de DoG es de 1.85 veces más rápido en el GPU, en las secciones de detección y eliminación de Puntos Característicos(PC) se encuentra la mayor aceleración, siendo de hasta 40.66 veces más, las anteriores aceleraciones son promedio entre la de la imagen del gato y el castor por que no eran cifras tan lejanas, pero en la parte de la orientación podemos notar como con imágenes más grandes es más rápido procesarlas en el GPU, teniendo una aceleración de 8.10 veces para el gato y solo una aceleración de 1.40 para el castor.

Lo que se hizo después, fue medir el tiempo en que se tardaban en obtener los puntos característicos 3 diferentes implementaciones, la primera la que desarrolle para CUDA, la de OpenSIFT y por último la que se encuentra en OpenCV. Se puede ver los resultados obtenidos en la tabla 5-3.

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

Tabla 5-3: Las unidades de tiempo son en milisegundos y la resolución en pixeles

Pero estas imágenes no eran las más adecuadas para hacer pruebas, ya que el robot de servicio Justina trabaja con imágenes como las de las figuras 5-3, 5-4, 5-5 y 5-6. Las primeras tres son objetos que tiene que manipular, la figura 5-6 es donde tiene que localizar los objetos. Lo que se hizo fue medir cuento tiempo se tardaban en generar los puntos característicos las 3 implementaciones anteriormente mencionadas y cambiar las resoluciones de estas imágenes. Los resultados los podemos ver a continuación en las siguientes tablas:



Figura 5-3: Puntos característicos encontrados en la imagen del stevia

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

Tabla 5-4: Las unidades de tiempo son en milisegundos y la resolución en pixeles



Figura 5-4: Puntos característicos encontrados en la imagen del café

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

Tabla 5-5: Las unidades de tiempo son en milisegundos y la resolución en pixeles



Figura 5-5: Puntos característicos encontrados en la imagen de la sopa

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

Tabla 5-6: Las unidades de tiempo son en milisegundos y la resolución en pixeles



Figura 5-6: Puntos característicos encontrados en la imagen del estante

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

Tabla 5-7: Las unidades de tiempo son en milisegundos y la resolución en pixeles

Un factor que se creyó afectaría el tiempo de ejecución de todos los programas fue la cantidad de puntos característicos que existen en la imagen, cosa que no paso, Se puede observar cómo se repite el fenómeno que notamos con el castor y el gato. Entre más grande es la imagen obtenemos una aceleración 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 pruebas a las que es sometido el robot de servicio Justina son más rigurosas, por este motivo los sistemas deben ser cada vez más rápidos y confiables. También las nuevas generaciones de sensores son mucho mejores tenemos más información para analizar, actualmente se utiliza un sensor Kinect para obtener las imágenes, esta tiene una resolución de 640 x 480 px y se utiliza el código de openSIFT para procesar esta imágenes, después se adquirió una cámara RGB Logitech C920 que tiene una de 1920 x 1080 px, el procesamiento de la imagen fue mucho más lento. Se planea empezar a usar también el nuevo sensor de Microsoft Kinect 2 el cual también tiene una resolución de 1920 x 1080 px.

Los resultados obtenidos tiene dos matices en mi opinion, donde las imágenes son muy pequeñas y hay poco que procesar , cuando el robot necesita encontrar objetos en una mesa, segmenta los objetos de la mesa y quedan solo los objetos, que no tienen una cantidad muy grande de pixeles a procesar, por lo que aun que es un poco más rápido procesarlos en el GPU, no tiene gran aceleración en comparación a la librería de OpenCV y es mucho más sencillo implementarlo; la otra parte es que se ha estado trabajando para que Justina no solo encuentre objetos en la mesa sino que debe

tener capacidad para encontrarlos en un libreo o estante. En el laboratorio de BioRobotica de la UNAM se ha estado tratando de segmentar estos estantes o libreros para poder seguir con la misma metodología al momento de buscar objetos, cosa que no se ha logrado con éxito total. Lo que se implemento fue hacer una búsqueda de los objetos en toda la imagen, para esto se intentó hacer con las imágenes que entrega el Kinect, pero la baja calidad de la imagen no permitía encontrar los objetos, ya que la distancia de donde se tomaba la foto era más lejos de lo que acostumbra ser. Se cambió la fuente de la imagen por la cámara C920 y los resultados mejoraron, pero el tiempo de procesamiento aumento.

EL robot Justina participa en una competencia principalmente llamada Robocup, en 2015 se puso una nueva prueba llamada *Manipulation and object recognition* donde se tenía que buscar objetos en un librero y manipularlos básicamente. Para esta prueba solo dan 3 minutos para reconocer y manipular objetos, de los cuales nosotros gastamos todo el tiempo solo reconociendo, 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.

El punto a resaltar es que la cantidad de datos que se tienen que procesar crecieron solo tres veces y se tarda en procesar hasta 13 veces más tiempo que con las anteriores resoluciones, lo que empieza a sonar como una buena idea es voltear a ver a las tarjetas gráficas de propósito general, las cuales ya se cuentan con ellas en las laptops que el robot usa. Observando los resultados con resoluciones de 1920 x 1080 podemos ver una aceleración de hasta 5 veces respecto a OpenSIFT y 2 veces contra la librería OpenCV, ya que en lugar de que el proceso tarde 5 segundos por foto estaría tardando solo un segundo.

6.2. Trabajo a Futuro

Como se vio hay nuevas necesidades para este Robot Justina, y con un aceleramiento de 5 veces no me parece suficiente por lo que hay que seguir trabajando en esto. Hay que hacer pruebas en diferentes tarjetas gráficas más poderosas y de nueva generación.

6.2. TRABAJO A FUTURO

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. Otro aspecto que se podría mejorar es al momento de lanzar el Kernel ya que hay que buscar cual es la configuración adecuada para el lanzamiento, con esto me refiero al número de bloques e hilos que ejecutaran el programa.

Encontrar otra forma más rápida de pasar los datos de la memoria RAM a la memoria de la GPU ya que ese es un cuello de botella importante, encontrar otra forma de manejar las imágenes para que los accesos a memoria será más rápidos. En fin tratar de sacarle todo el jugo a la tarjeta gráfica adaptando el código a esta.

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, no sé si usar las tarjetas gráficas para hacer el emparejamiento sea lo óptimo pero igual se puede intentar y ver si arroja resultados favorables.

APÉNDICE A

KERNEL CONVOLUCIÓN

```
--global__ void Convolution(float* image, float* mask,
                           ArrayImage* PyDoG, int maskR, int maskC,
                           int imgR, int imgC, float* imgOut, int idxPyDoG)

{
    int tid= threadIdx.x;
    int bid= blockIdx.x;
    int bDim=blockDim.x;
    int gDim=gridDim.x;
    int iImg=0;
    float aux=0;

    int pxlThrd = ceil((double)(imgC*imgR)/(gDim*bDim));
    for(int i = 0; i <pxlThrd; ++i)

    {
        iImg=(tid+(bDim*bid)) + (i*gDim*bDim);
        if(iImg < imgC*imgR){

            int condition=maskC/2+imgC*(floor((double)maskC/2));
            if (iImg-condition < 0 ||
```

```

iImg+condition > imgC*imgR ||
iImg %imgC < maskC/2 ||
iImg %imgC > (imgC-1)-(maskC/2) )

{
    aux=0;

}else{

    int itMask = 0;

    int itImg=iImg-condition;

    for (int j = 0; j < maskR; ++j)

    {
        for (int h = 0; h < maskC; ++h)

        {

            aux+=image[itImg]*mask[itMask];

            ++itMask;

            ++itImg;

        }

        itImg+=imgC-maskC;

    }

    imgOut[iImg]=aux;

    aux=0;

}

}

PyDoG[idxPyDoG].image=imgOut;

}

```

APÉNDICE B

KERNEL LOCALIZACIÓN DE MÁXIMOS Y MÍNIMOS

```
--global__ void LocateMaxMin(ArrayImage* PyDoG , int idxPyDoG ,
                           float * imgOut ,MinMax * mM , int maskC , int imgR ,
                           int imgC , int idxmM)

{
    int tid= threadIdx.x;
    int bid= blockIdx.x;
    int bDim=blockDim.x;
    int gDim=gridDim.x;
    int iImg=0;
    int pxlThrd = ceil((double)(imgC*imgR)/(gDim*bDim));
    for(int i = 0; i <pxlThrd; ++i)
    {
        int min=0;
        int max=0;
        float value=0.0;
        float compare =0.0;
        iImg=(tid+(bDim*bid)) + (i*gDim*bDim);
        if(iImg < imgC*imgR){
```

```

int condition=maskC/2+imgC*(floor((double)maskC/2));

if (iImg-condition < 0 ||

iImg+condition > imgC*imgR ||

iImg%imgC < maskC/2 ||

iImg%imgC > (imgC-1)-(maskC/2) )

{

imgOut[iImg]=0;

}

else{

value=PyDoG[idxPyDoG].image[iImg];

for (int m = -1; m < 2; ++m)

{

int itImg=iImg-(1+imgC);

for (int j = 0; j < 3; ++j)

{

for (int h = 0; h < 3; ++h)

{



compare =PyDoG[idxPyDoG+m].image[itImg];

if(value<=compare && max==0)

{



++min;

}

else if(value>=compare && min==0)

{



++max;

}

++itImg;

}

itImg+=imgC-3;
}

```

```
    }

}

if( (min==26 || max==26)) {

    imgOut[iImg]=1;

} else{

    imgOut[iImg]=0;

}

}

}

mM[idxmM].minMax=imgOut;

}
```

APÉNDICE C

KERNEL REMOVER PUNTOS MALOS

```
--global__ void RemoveOutlier(ArrayImage* PyDoG , MinMax * mM,
                                int idxmM , int idxPyDoG , int imgR,int imgC ,float* auxOut)
{
    int tid= threadIdx.x;
    int bid= blockIdx.x;
    int bDim=blockDim.x;
    int gDim=gridDim.x;
    int iImg=0;
    int pxlThrd = ceil((double)(imgC*imgR)/(gDim*bDim));
    for(int i = 0; i <pxlThrd; ++i
    {
        iImg=(tid+(bDim*bid)) + (i*gDim*bDim);

        if(iImg < imgC*imgR){
            if(mM[idxmM].minMax[iImg]>0 && PyDoG[idxPyDoG].image[iImg]>0.05)
            {
                float d, dxx, dyy, dxy, tr, det;
```

```

d = PyDoG[idxPyDoG].image[iImg];
dxx = PyDoG[idxPyDoG].image[iImg+1] +
       PyDoG[idxPyDoG].image[iImg-1] - (2*d);
dyy = PyDoG[idxPyDoG].image[iImg+imgC] +
       PyDoG[idxPyDoG].image[iImg-imgC] - (2*d);
dxy = (PyDoG[idxPyDoG].image[iImg+1+imgC] -
        PyDoG[idxPyDoG].image[iImg-1+imgC] -
        PyDoG[idxPyDoG].image[iImg+1-imgC] +
        PyDoG[idxPyDoG].image[iImg-1-imgC])/4.0;

tr = dxx + dyy;
det = dxx*dyy - dxy*dxy;

if(det<=0 && !(tr*tr/det < 12.1))
    mM[idxmM].minMax[iImg]=0;
} else
{
    mM[idxmM].minMax[iImg]=0;
}
auxOut[iImg]=mM[idxmM].minMax[iImg];
}

}

}

```

APÉNDICE D

KERNEL ASIGNAR MAGNITUD Y ORIENTACIÓN

```
--global__ void OriMag(ArrayImage* PyDoG, int idxPyDoG,
                      int imgR,int imgC , ArrayImage* Mag, ArrayImage* Ori,
                      int idxMagOri, float* MagAux, float* OriAux)

{
    int tid= threadIdx.x;
    int bid= blockIdx.x;
    int bDim=blockDim.x;
    int gDim=gridDim.x;
    float dx,dy;
    int iImg=0;
    int pxlThrd = ceil((double)(imgC*imgR)/(gDim*bDim));
    for(int i = 0; i <pxlThrd; ++i)
    {
        iImg=(tid+(bDim*bid)) + (i*gDim*bDim);
        if(iImg < imgC*imgR){
            int condition=1/2+imgC*(floor((double)1/2));
            if (iImg-condition < 0 || iImg+condition > imgC*imgR ||
                iImg%2==0)
                OriAux[iImg]=0;
            else
                OriAux[iImg]=1;
        }
    }
}
```

```

iImg %imgC < 1/2 ||
iImg %imgC > (imgC-1)-(1/2) )

{
    OriAux [iImg]=0;
    MagAux [iImg]=0;
}

else{
    dx=PyDoG [idxPyDoG].image [iImg+1] -
        PyDoG [idxPyDoG].image [iImg-1];
    dy=PyDoG [idxPyDoG].image [iImg+imgC] -
        PyDoG [idxPyDoG].image [iImg-imgC];
    MagAux [iImg]=sqrt (dx*dx + dy*dy);
    OriAux [iImg]=atan2 (dy , dx);
}

}

}

Mag [idxMagOri].image= MagAux;
Ori [idxMagOri].image= OriAux;

}

```

APÉNDICE E

KERNEL GENERAR PUNTOS CARACTERÍSTICOS

```
--global__ void KeyPoints(ArrayImage * Mag ,  
                         ArrayImage * Ori, MinMax * mM , int idxM0mM ,  
                         keyPoint * KP, float sigma, int imgR,int imgC ,  
                         int octava )  
{  
    int tid= threadIdx.x;  
    int bid= blockIdx.x;  
    int bDim=blockDim.x;  
    int gDim=gridDim.x;  
    float o = 0;  
    int x=0, y=0, octv=-1;  
    int iImg=0;  
    int pxlThrd = ceil((double)(imgC*imgR)/(gDim*bDim));  
    for(int i = 0; i <pxlThrd; ++i)  
    {  
        iImg=(tid+(bDim*bid)) + (i*gDim*bDim);  
        octv=-1;  
        if(iImg < imgC*imgR ){
```

```

if(mM[idxM0mM].minMax[iImg]>0){

    int condition=9/2+imgC*(floor((double)9/2));

    if (iImg-condition < 0 ||

        iImg%imgC > (imgC-1)-(9/2) )

    {

        o=-1.0;

        x=-1;

        y=-1;

        octv=-1;

    }

    else{

        float histo[36]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

        octv=octava;

        x=iImg%imgC;

        y=iImg/imgC;

        int idxM0= (iImg-4)-(4*imgC);

        float exp_denom = 2.0 * sigma * sigma;

        float w;

        int bin;

        for (int i = -4; i < 5; ++i)

        {

            for (int j = -4; j < 5; ++j)

            {

                w = exp( -( i*i + j*j ) / exp_denom );

                bin=round((double)(36*Ori[idxM0mM].image[idxM0])

                           /6.28318530718);

                bin = ( bin < 36 )? bin : 0;

                histo[bin]= w*Mag[idxM0mM].image[idxM0];
}

```

```

        ++idxM0;

    }

    idxM0=idxM0+imgC-9;

}

int idxH=0;

float valMaxH = histo[0];

for (int i = 1; i < 36; ++i)

{

    if(histo[i]>valMaxH){

        idxH = i;

    }

}

int l = (idxH == 0)? 35:idxH-1;

int r = (idxH+1)%36;

float bin_= bin + ((0.5*(histo[l]-histo[r]))/

                    (histo[l]-(2*histo[idxH])+histo[r]));

bin_= ( bin_ < 0 )? 36 + bin_ :

( bin_ >= 36 )? bin_ - 36 :

bin_;

o=((6.28318530718*bin_)/36)-3.141592654;

}

}

KP[iImg].orientacion=o;

KP[iImg].x=x;

KP[iImg].y=y;

KP[iImg].octv=octv;

}

}

}

```



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] NVIDIA. Cuda c programming guide [en linea]. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3yegx1PRY>, Accedido: [25/08/2015].
- [4] David Kirk and Wen-mei Hwu. *Programming massively parallel processors : a hands-on approach*. Morgan Kaufmann, San Francisco, CA, USA, 2010.
- [5] Jack Dongarra. Lo que la gente dice[en linea]. 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] I Buck. Brook : A Streaming Programming Language. *Communication*, (October):1–12, 2001.
- [8] NVIDIA. Language solutions [en linea]. <https://developer.nvidia.com/language-solutions>, Accedido: [25/08/2015].

- [9] Whitepaper Nvidia, Next Generation, and Cuda Compute. Whitepaper NVIDIA's Next Generation CUDA Compute Architecture. *ReVision*, 23(6):1–22, 2009.
- [10] Nvidia. Kepler GK110. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*, 2012.
- [11] Nvidia Geforce Gtx, Featuring Maxwell, The Most, Advanced Gpu, and Ever Made. NVIDIA GeForce GTX 980. pages 1–32.
- [12] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. EBL-Schweitzer. Wiley, 2014.
- [13] OpenCV. Opencv documentation [en linea]. <http://docs.opencv.org/2.4/index.html>, Accedido: [25/08/2015].
- [14] Rob Hess. Opensift [en linea]. [https://robwhess.github.io/opensift/1](https://robwhess.github.io/opensift/), Accedido: [25/08/2015].

ÍNDICE DE FIGURAS

2-1. Espacio Escala de Diferencia de Gaussianas	7
2-2. Espacio Escala de Diferencia de Gaussianas	8
2-3. Histograma de Orientación	11
2-4. Descriptor	13
2-5. Comparación entre GPU vs CPU [3]	14
3-1. Sistema Híbrido	16
3-2. SIMD	19
3-3. La arquitectura Fermi tiene sus 16 SM alrededor de la memoria compartida L2 cache [9]	20
3-4. Fermi Streaming Multiprocessor (SM)[9]	21
3-5. Kepler Next Generation Streaming Multiprocessor (SMX) [10]	23
3-6. Maxwell Streaming Multiprocessor (SMM) [11]	26
3-7. Organización de bloques e hilos [3]	28
3-8. Asignación de bloques por SM [3]	30
3-9. Tipos de Memoria [12]	31
3-10. Programación Heterogénea [3]	33

4-1. Todos los procesos son partes diferentes del algoritmo	38
4-2. Todos los procesos tienen la misma especificación	39
4-3. División del algoritmo SIFT a paralelizar	40
4-4. Proceso general de los kernels	42
4-5. Lanzamiento de Kernels	42
4-6. Esta es la imagen de entrada que arrojaron como resultado las de las figuras 4-8, 4-9, y 4-11	43
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) $\sigma_4 = 2.16 - \sigma_5 = 2.72$	43
4-8. Espacio Escala de Diferencia de Gaussianas	44
4-9. Mascara. Búsqueda con las imágenes del espacio escala: a) 0,1,2 ; b) 1,2,3; c)2,3,4 . .	46
4-10. Mascara Filtrada de la búsqueda con las imágenes del espacio escala: a) 0,1,2 ; b) 1,2,3; c)2,3,4	48
4-11. Puntos característicos	50
5-1. Puntos característicos encontrados en la imagen del castor	52
5-2. Puntos característicos encontrados en la imagen del gato	52
5-3. Puntos característicos encontrados en la imagen del stevia	54
5-4. Puntos característicos encontrados en la imagen del café	55
5-5. Puntos característicos encontrados en la imagen de la sopa	56
5-6. Puntos característicos encontrados en la imagen del estante	57