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 Ciencias de la 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 Ciencias de la 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

CAPÍTULO 1

INTRODUCCIÓN

1.1. Contexto

Los robots móviles rigen su comportamiento en base a el software, dependiendo de cuanta interacción tenga el robot con su entorno, podría llegar a ser mas complejo que todo el complejo 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.

El sistema de visión humano, es al que más recurre para obtener información de su entorno. Por ello no es de extrañarse que la visión computacional, en la robótica, tenga una participación muy importante,por que estas maquinas 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 es preciado en cualquier rama y esta no es la excepción, la forma de ganar tiempo que se a venido sesgando por hardware es el paralelismo, y no solo hablo de procesadores multinúcleo, las tarjetas gráficas se pueden programar para realizar tareas de propósito general.

1.2. Problema a resolver

Los algoritmos que se utilizan, en visión computacional muchas veces son muy confiables, pero consumen mucho tiempo de procesador, por esto se ha tratado de hacer mas 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 lo anteriormente dicho, 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 mas 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 las tarjetas gráficas, usando computo heterogéneo, tener un mejor desempeño, en cuanto al tiempo en el que se ejecutan, de algoritmos de visión computacional, que ejecutándose de manera secuencial son robustos pero lentos"

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, obtenida con las cámaras montadas en un robot móvil. El desempeño se medirá comparando el tiempo que se obtuvo, con el sistema actual del robot y el que se propone.

1.4. Estructura de la tesis

Capitulo 1: Introducción. En este capitulo se presenta de lo que tratara en general este trabajo de tesis, planteando el contexto en el que se trabaja, el problema a resolver, la hipótesis y cual seria el alcance de este trabajo.

Capitulo 2: Algoritmos de extracción y descripción de características. En este capitulo 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.

Capitulo 3: Computo en GPU's. Este capitulo tratara de como ha cambiado, los procesadores multi-núcleo, la forma en la que programamos y sobre todo el computo en cooperación con las tarjetas gráficas. Nos enfocaremos en las GPU de la familia de Nvidia, se vera 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.

Capitulo 4: SIFT en GPU.

Capitulo 5: Resultado y Conclusiones.

Capitulo 6: Trabajo a Futuro.

CAPÍTULO 2

ALGORITMOS DE EXTRACCIÓN Y DESCRIPCIÓN DE CARACTERÍSTICAS

Las características de los objetos son cualidades que nos servirán para identificarlos dentro de una imagen donde pueden existir objetos similares. Para poder discernir de los objetos que estén en la 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 toda 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. Y para generar un descriptor, es la misma situación, dependerá del algoritmo. En este capítulo se explicarán un algoritmo para extraer características y generar su descriptor.

2.1. Scale-Invariant Feature Transform SIFT

El algoritmo de SIFT, propuesto por Lowe en [**?**], 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:

2.1.1. 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

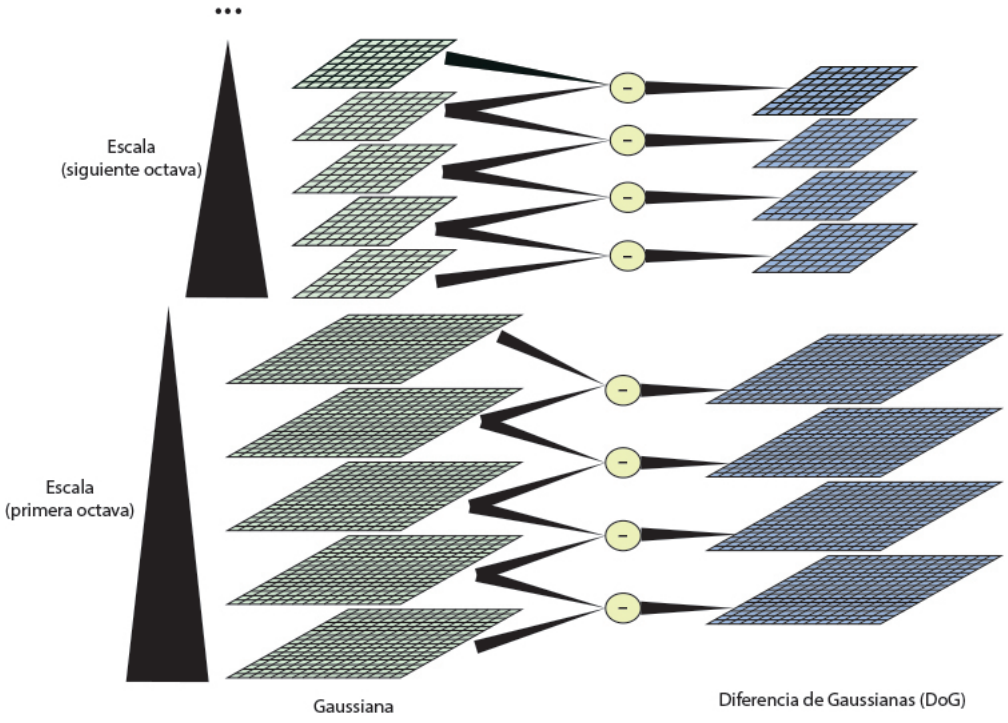

Figura 2-1: Espacio Escala de Diferencia de Gaussianas

de gaussianas 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:

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

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

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 sera 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 mascaras 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 emborronadas en la pila, por octava. Para extraer las

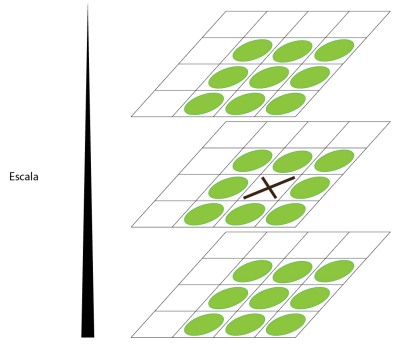

Figura 2-2: Espacio Escala de Diferencia de Gaussianas

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.

2.1.2. 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 debemos obtener la serie de Taylor del espacio escala $D(x, y, \sigma)$:

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

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 evaluara al punto extremo sera, $D(\hat{X})$, la cual rechazara al punto si es de muy bajo contraste, la cual se obtiene de sustituir $\hat{X}$ en $D(X)$:

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

En el trabajo de Lowe [**?**], se puede ver que 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 [**?**], se puede evitar el calculo 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 mas 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 [**?**] se encontró un valor experimental para $r = 10$, que elimina los puntos extremos que tengan la razón entre las dos curvas mayor que 10.

2.1.3. 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 sera invariante a la rotación. La ubicación en el espacio escala del punto característico, es usada para seleccionar la imagen suavizada por una mascara 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.

Para cada muestra agregada se ponderada por la magnitud de su gradiente y por una mascara circular gaussiana ponderada con σ, que es 1.5 veces que de la ubicación 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 mas 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 si, para interpolar la posición del pico con mas precisión.

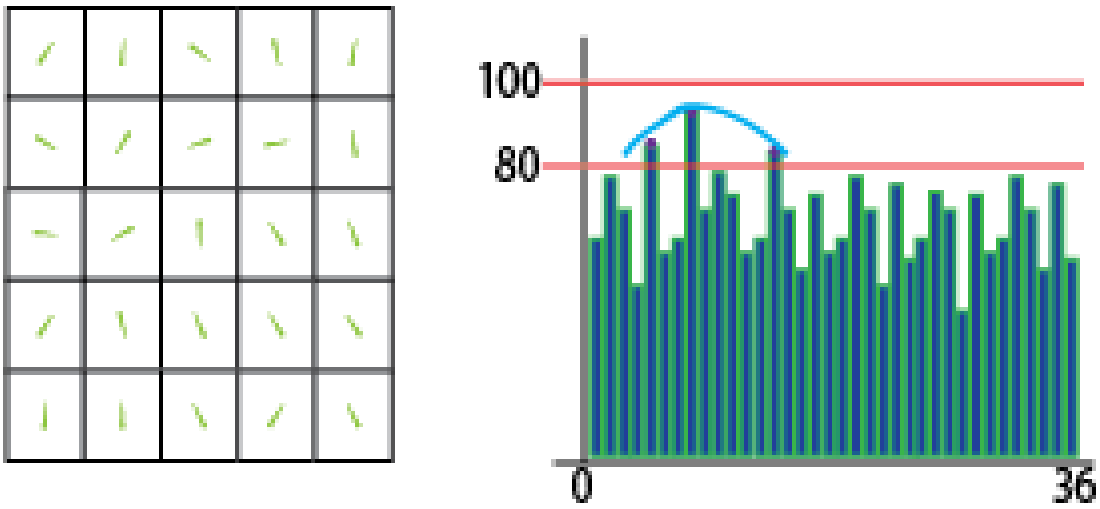

Figura 2-3: Histograma de Orientación

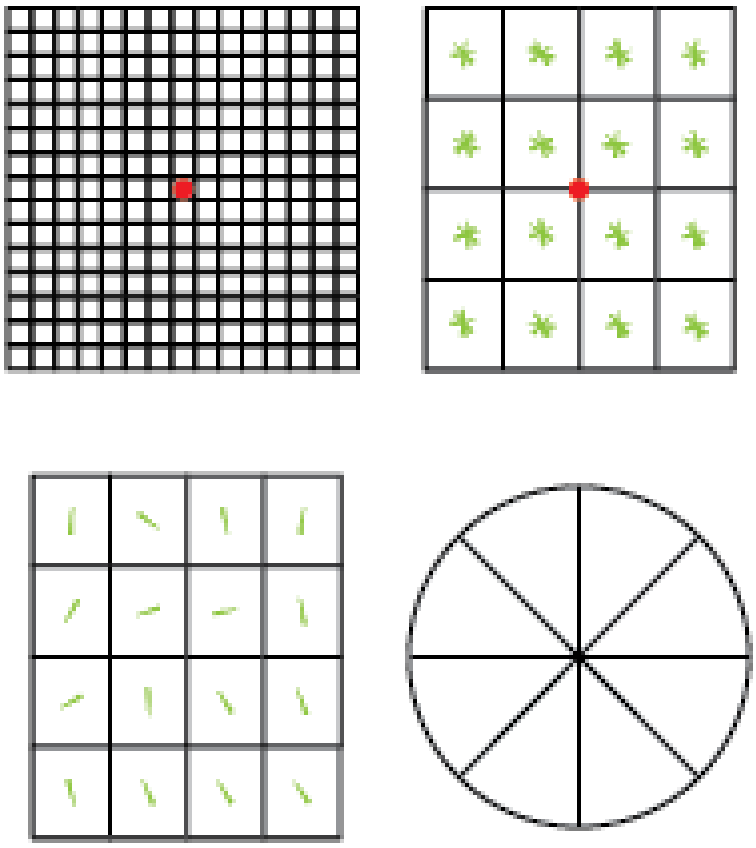

Figura 2-4: Descriptor

2.1.4. Descriptor de puntos característicos

Hasta este momento tenemos 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 sera de $4 \times 4 \times 8 = 128$ elementos.

CAPÍTULO 3

COMPUTO EN GPU'S

Hasta hace 12 años la velocidad a la que crecían 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, por que la ejecución de estas son comprensibles paso a paso, mediante el código. Pero un programa secuencial ejecutándose en un solo núcleo del procesador, no sera 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 mas rápido. Existen dos corrientes principales en cuanto a los procesadores multi-núcleo, el primero, es donde se pretende mantener la velocidad de los programas secuenciales, mientras se mueven entre múltiples núcleos; la segunda, se centra mas en la ejecución de aplicaciones en paralelo, tiene un gran numero 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 ingles GPU.[**?**]

3.1. GP-GPUs Nvidia

"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 tándem con CPU de múltiples núcleos".[?]

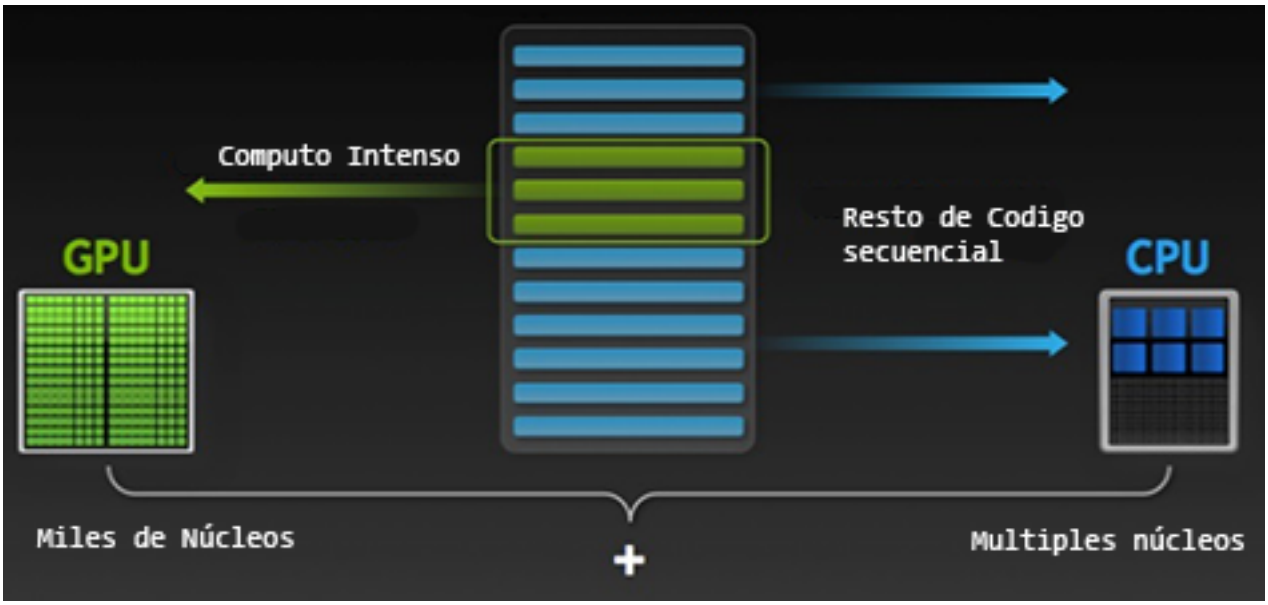

Figura 3-1: Sistema Híbrido

3.1.1. Breve Historia

La necesidad de mejores gráficos para los vídeo juegos, 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 ingles 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[?] 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 llamo la atención de otros profesionales, que se integraron a el trabajo de los artistas y desarrolladores de vídeo 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 memorizaras 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* [**?**], 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 mas fácil de manejar, ya que era de más alto nivel, y lo mas importante los programas escritos en *Brook* eran hasta 7 veces mas rápido que códigos similares existentes.

La compañía NVIDIA se dio cuenta que tenia 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 seria 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. CUDA

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 computo, 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, multiples datos* (SIMD).

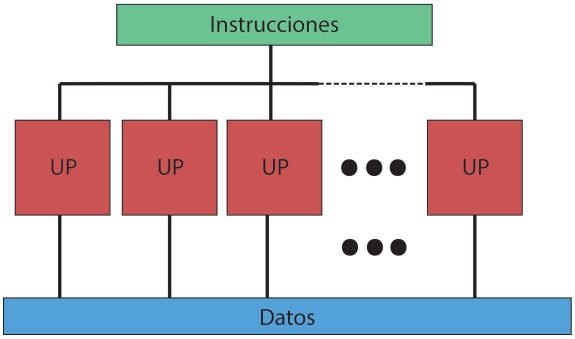

Figura 3-2: SIMD

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[**?**].

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

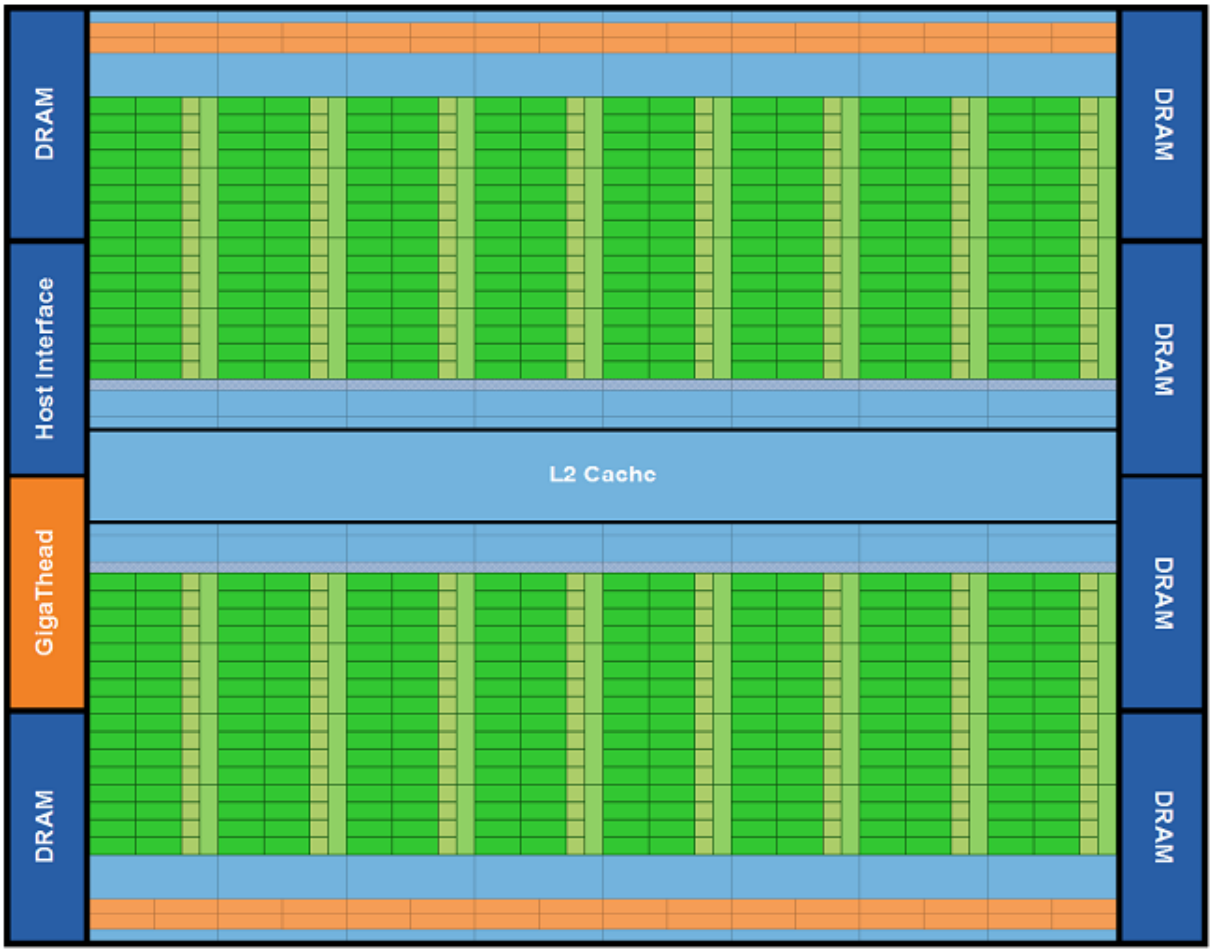

Figura 3-3: La arquitectura Fermi tiene sus 16 SM al rededor de la memoria compartida L2 cache

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 implemento 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.

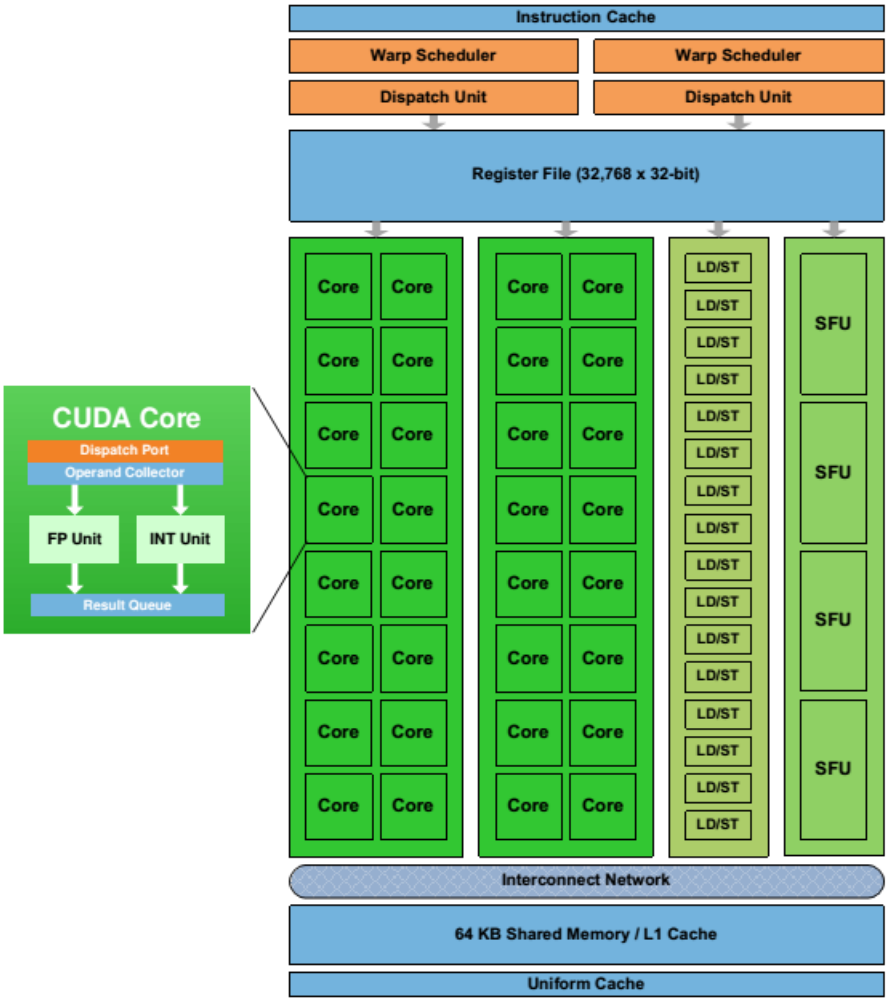

Figura 3-4: Fermi Streaming Multiprocessor (SM)

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 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. Tambien 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 mas complejas como senos, cosenos, reciproco y raíz cuadrada.

Kepler

La arquitectura Kepler [?], modifico los SM de su antecesor llamándolo Next Generation Streaming Multiprocessor (SMX), es el nuevo procesador de esta arquitectura, en la cual encontraremos que esta 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 de el 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)

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 para programas donde se requiere probar o filtrar datos de una imagen, en esta arquitectura dejo de ser un hardware dedicado solo a este objetivo, se dejo 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 agrego una característica, donde no se requiere de el CPU para lanzar programas en la GPU, lo que significa que el GPU tiene la capacidad de generar mas carga de trabajo, administrar recursos y obtener resultados dentro de la misma GPU, en la zona de mas interés, donde se pueda requerir más poder de computo.

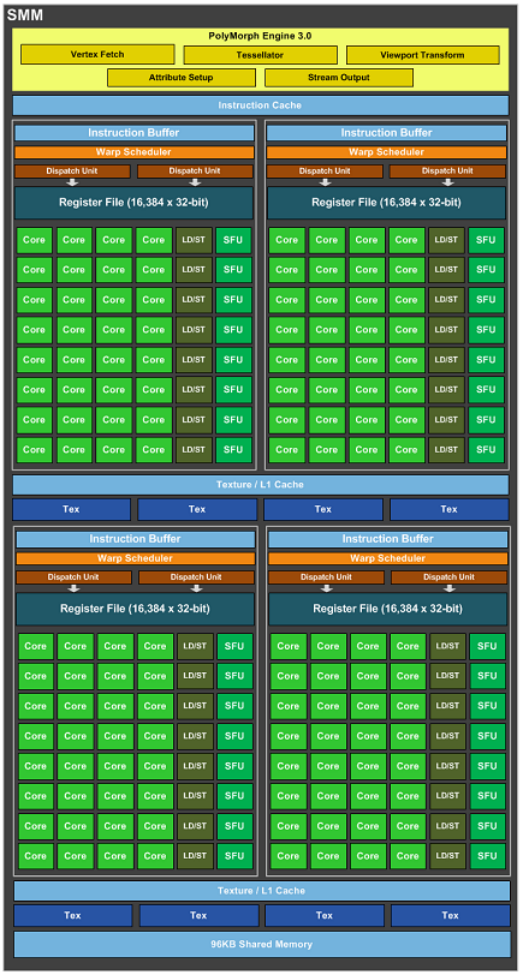

Figura 3-6: Maxwell Streaming Multiprocessor (SMM)

Maxwell

La arquitectura Maxwell[**?**], tuvo un cambio en su diseño para proporcionar un cambio dramático en su desempeño. Lo que genero este desempeño fue el nuevo diseño que le dieron a los nuevos SM llamados SM Maxwell (SMM). El numero de CUDA cores bajo 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 mas eficiente el espacio y la energia gastada para el manejo de la transferencia de datos.

La memoria compartida incremento a 96KB, la cual ya no se comparte con la memoria cache L1, ahora la memoria de textura comparte espacio con la cahce L1. Los registros, las SFU, y las unidades load/store siguen siendo la misma cantidad.

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 mas 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      }
```

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.

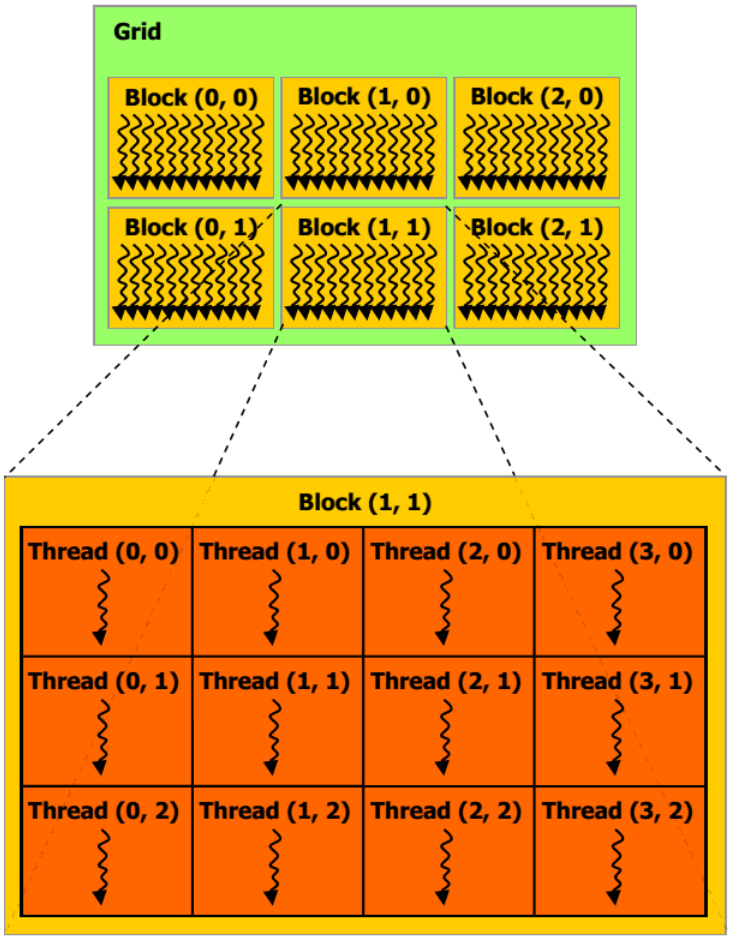

Figura 3-7: Organización de bloques e hilos

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 tiene un

identificador al cual se puede acceder por la variable **blockIdx**, existe otra variable importante y es la que nos dará la dimension del bloque, se llama **blockDim**. A continuación veremos un ejemplo de como se lanzaría 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          }
```

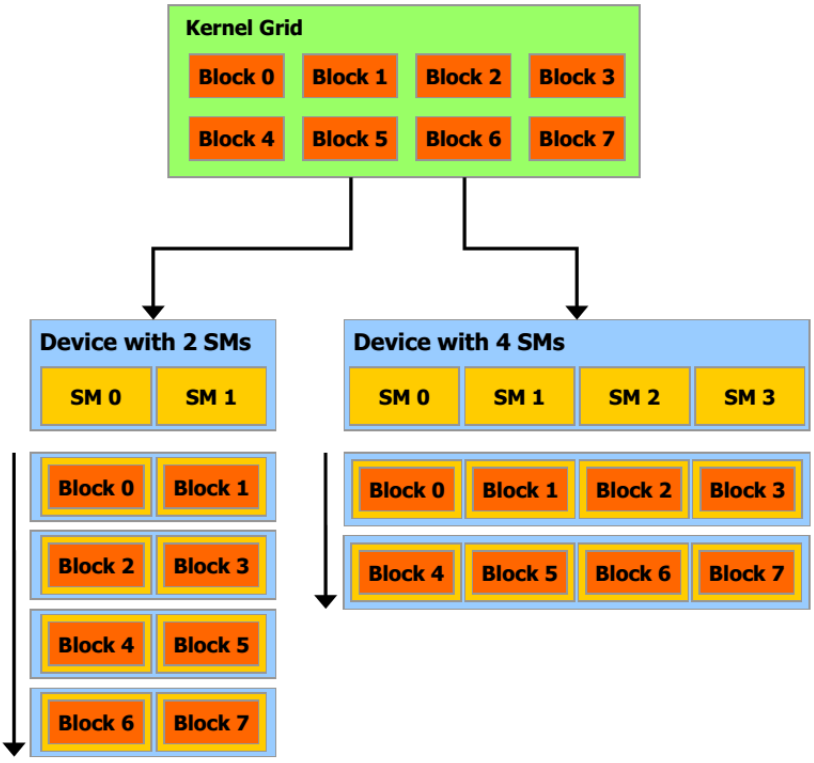

Figura 3-8: Asignación de bloques por SM

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 esperaran 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 como 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 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.

Los *registros* son el tipo de memoria con la lectura/escritura mas 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 bloque, esto genera que los hilos de el 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.

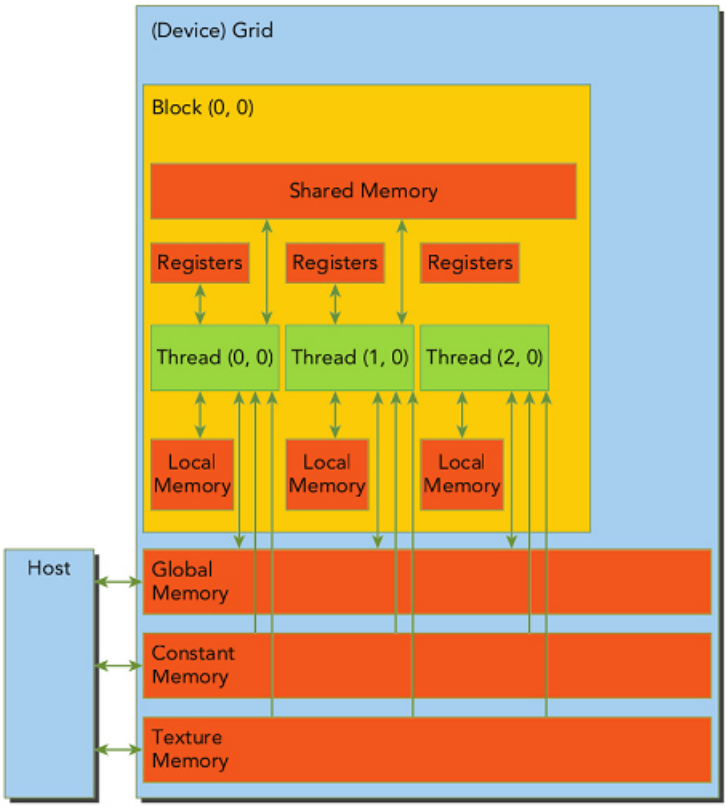

Figura 3-9: Tipos de Memoria

La *memoria constante* es una memoria de solo lectura, en la cual albergaremos datos que no cambiaran a lo lago 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 mas 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, esta 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 gygabytes. 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 son persistentes, 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 mas 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

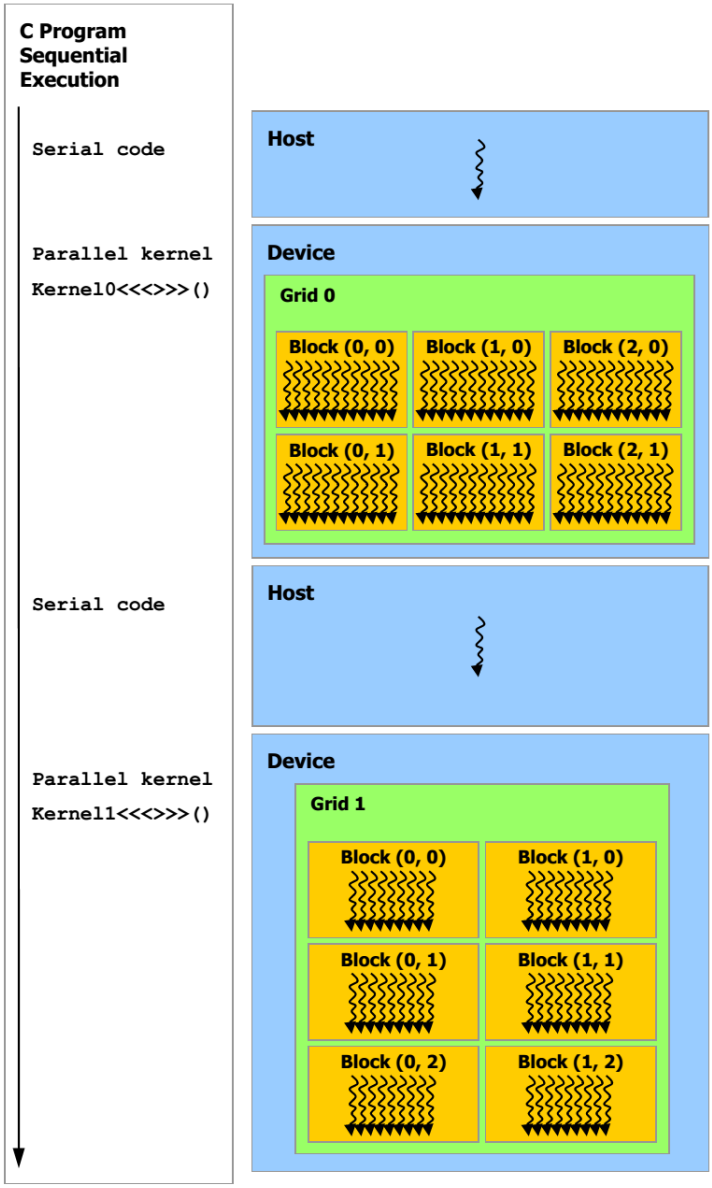

Figura 3-10: Programación Heterogénea

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 cuando necesitara usar al dispositivo (Device) GPU, el anfitrion 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 desde algún kernel se definiran 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 como 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. La final solo quedan liberar los recurso 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                 L* datosD;
14                 cudaMalloc(&datosD,size);
15                 ...
16                 cudaMemcpy(datosD,src,size,cudaMemcpyHostToDevice);
17                 ...
18                 dim3 gridDim(...,...,...);
```

```
19                  dim3 blockDim(...,...,...);
20                  KernelUno<<<gridDim,blockDim>>>(datosD,...);
21                  ...
22                  KernelDos<<<...,...>>>(...);
23                  ...
24                  cudaMemcpy(res,datosD,size,cudaMemcpyHostToDevice);
25                  ...
26                  cudaFree(datosD);
27                  ...
28          }
```

CAPÍTULO 4

SIFT EN GPU

La correcta paralelización de un algoritmo no es nada trivial. Después de el capitulo anterior, al ver todas las ventajas que tenemos en los GPU‘s 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 paralelizar. Para saber como analizar si un algoritmo es paralelizable primero debemos dar una definición de que es un programa paralelo:

”Programa paralelo es la especificación de dos o mas procesos simultáneos que cooperan entre si con un fin en común”

Podemos sacar 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 tenemos que cambiar la forma en que programamos, ahora no solo pensaremos como llegar a un objetivo paso a paso, sino que debemos pensar como muchos procesos trabajaran juntos para alcanzar un objetivo, esto inmediatamente me da la idea de repartir o dividir el trabajo entre todos ellos. Así que para realizar la labor de paralelizar el algoritmo debemos analizar básicamente tres casos de paralelismo:

- *Funcional*: Aquí lo que se divide es el algoritmo, buscamos pasos en el algoritmo que no dependan de otra parte del mismo y los ponemos a ejecutarse simultáneamente en diferentes procesos. Requiere de sincronizan muy cuidadosamente para que las diferentes partes de el algoritmo no interfieran entre si.

- *Dominio*: Se repartirán los datos en los múltiples procesos los cuales tienen una especificación idéntica. El sincronizan 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.

Ahora que tenemos el algoritmo y las herramienta para mejorar su rendimiento por medio de la paraleización. Veremos como analizamos 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

Primero dividiremos en 6 partes el algoritmo de SIFT, estas partes no se ejecutaran simultáneamente, es solo que el algoritmo es bastante largo y al separarlo en estas partes podemos simplificar en diferentes kerneles que tendrán una secuencia de ejecución. Lo que quiere decir que lo que tendremos que repartir entre los múltiples procesos serán los datos de entrada, con lo cual estaremos en la categoría de paralelismo de *dominio*.

Ahora

4.2. Implementación

CAPÍTULO 5

RESULTADOS Y CONCLUSIONES

CAPÍTULO 6

TRABAJO A FUTURO

ÍNDICE DE FIGURAS