UAH

Optimización de funciones de alto coste computacional mediante su implementación en GPU

Máster Universitario en Sistemas Electrónicos Avanzados. Sistemas Inteligentes

Departamento de Electrónica

Presentado por:

D. Francisco José Nombela Blanco

Dirigido por:

Dra. Cristina Losada Gutiérrez

D. David Jiménez Cabello

Alcalá de Henares, a 26 de Junio de 2014

Dra. Cristina Losada Gutiérrez

Director/TUTOR del Trabajo Fin de Máster de título: "Optimización de funciones de alto coste computacional mediante su implementación en GPU"

Realizado por D. Francisco José Nombela Blanco

Por la presente da su conformidad para que el citado trabajo sea presentado para su defensa.

Alcalá de Henares, a 25 de Junio de 2014

Firmado:

CRISTINA LOSADA

....

Contenido

Capítulo 1. Introducción....3
1.1. Objetivos 4
1.2. Estructura de la memoria....5
Capítulo 2. Fundamento teórico....7
2.1. Cámaras de profundidad basadas en tiempo de vuelo 7
2.1.1. Principio de funcionamiento 8
2.1.2. Errores asociados a cámaras *ToF* 10
2.2. Cámara SR-4000 14
2.3. Corrección del MpI 17
2.4. Arquitectura de cálculo paralelo en GPU 20
2.4.1. Arquitectura general. 21
2.4.2. Modelo de programación. 22
2.4.3. Modelo de memoria....24
2.4.4. Instrucciones comunes 27
2.4.5. Estructura del programa....29
2.4.6. Maximización de la ocupación de los multiprocesadores....31
2.4.7. CUDA Occupancy Calculator 33
2.4.8. Ventajas. 34
Capítulo 3. Algoritmo de corrección del MpI 37
3.1. Introducción 37
3.2. Diagrama de la propuesta de corrección MpI 37
3.3. Condiciones iniciales....38
3.4. Cálculo de las normales....39
3.4.1. Obtener núcleo de vecindad 39
3.4.2. Obtener ecuación de plano 42
3.4.3. Resumen de tiempos de cómputo de la función 47
3.5. Cálculo de los diferenciales de área....47
3.5.1. Resumen de tiempos de cómputo de la función 51
3.6. Cálculo de *rho* relativo 51

3.6.1. Obtener *rho* 51
3.6.2. Cálculo del máximo de un conjunto de valores en paralelo 55
3.6.3. Cálculo de *rho* relativo 61
3.6.4. Resumen de tiempos de cómputo de la función 63
Capítulo 4. Resultados 65
4.1. Resultados experimentales 65
4.1.1. Efecto del número de vecinos en el cálculo de normales 65
4.1.2. Resultado final: cálculo de normales, diferencial de área y coeficiente de absorción relativo (rho) 67
4.2. Análisis temporal 74
4.2.1. Tiempos medios de cada una de las funciones 74
4.2.2. Comparativa con otros entornos de programación 79
Capítulo 5. Conclusiones y trabajo futuro 81
Anexo 1. Características del sistema 83
Bibliografía 87

Índice de figuras

FIGURA 2.1: CÁMARA *TOF* SR-4000 7
FIGURA 2.2: PRINCIPIO DE FUNCIONAMIENTO DE UNA CÁMARA *TOF* 8
FIGURA 2.3: EFECTO GENERADO POR LA PRESENCIA DE *MPL* 13
FIGURA 2.4. REGIONES DE LA CÁMARA 14
FIGURA 2.5: EFICIENCIA EN LA ADQUISICIÓN RESPECTO A LA DISTANCIA 15
FIGURA 2.6: SISTEMA DE COORDENADAS DE LA CÁMARA 15
FIGURA 2.7: DIAGRAMA DEL EFECTO DEL *MPI* 17
FIGURA 2.8: ELEMENTOS DE LA CAPA SOFTWARE EN EL MODELO CUDA 20
FIGURA 2.9: ARQUITECTURA DE UNA *GPU* 21
FIGURA 2.10: FASE DE EJECUCIÓN EN CUDA 22
FIGURA 2.11: MODELO DE EJECUCIÓN EN CUDA 23
FIGURA 2.12: MODELO DE MEMORIA EN CUDA 25
FIGURA 2.13: EJEMPLO DE *KERNEL*. 29
FIGURA 2.14: EJEMPLO DE INVOCACIÓN DE UN *KERNEL* 30
FIGURA 2.15: CUDA OCCUPANCY CALCULATOR 33
FIGURA 2.16: GRÁFICAS DE PREDICCIÓN DE LA OCUPACIÓN 34
FIGURA 3.1: DIAGRAMA GENERAL DEL ALGORITMO ITERATIVO PARA LA CORRECCIÓN *MPI* 38
FIGURA 3.2: IMPACTO AL VARIAR EL TAMAÑO DEL BLOQUE EN *KERNEL1* 40
FIGURA 3.3: IMPACTO AL VARIAR EL NÚMERO DE REGISTRO POR HILO EN *KERNEL1* 40
FIGURA 3.4: RESUMEN DE RECURSOS *KERNEL1* 41
FIGURA 3.5: OBTENCIÓN DE UN PLANO 42
FIGURA 3.6: IMPACTO AL VARIAR EL TAMAÑO DEL BLOQUE EN *KERNEL2* 45
FIGURA 3.7: IMPACTO AL VARIAR EL NÚMERO DE REGISTRO POR HILO EN *KERNEL2* 45
FIGURA 3.8: RESUMEN DE RECURSOS *KERNEL2* 46
FIGURA 3.9: DIFERENCIAL DE ÁREA 47
FIGURA 3.10: IMPACTO AL VARIAR EL NÚMERO DE REGISTRO POR HILO EN ÁREA 49
FIGURA 3.11: IMPACTO AL VARIAR EL TAMAÑO DEL BLOQUE EN ÁREA 49
FIGURA 3.12: RESUMEN DE RECURSOS ÁREA 51
FIGURA 3.13: TRAZADO DE RAYOS PARA OBTENER *RHO* 52
FIGURA 3.14: IMPACTO AL VARIAR EL NÚMERO DE REGISTRO POR HILO EN *RHO3* 53
FIGURA 3.15: IMPACTO AL VARIAR EL TAMAÑO DEL BLOQUE EN *RHO3* 53
FIGURA 3.16: RESUMEN DE RECURSOS *RHO3* 54
FIGURA 3.17: REDUCCIÓN PARALELA CON DIRECCIONAMIENTO SECUENCIAL 55
FIGURA 3.18: IMPACTO AL VARIAR EL TAMAÑO DEL BLOQUE EN *REDUCE6* 56
FIGURA 3.19: IMPACTO AL VARIAR EL NÚMERO DE REGISTRO POR HILO EN *REDUCE6* 57
FIGURA 3.20: IMPACTO AL VARIAR LA MEMORIA COMPARTIDA USADO POR BLOQUE EN *REDUCE6* 57
FIGURA 3.21: RESUMEN DE RECURSOS *REDUCE6* 58
FIGURA 3.22: IMPACTO AL VARIAR EL TAMAÑO DEL BLOQUE EN *REDUCE3* 59
FIGURA 3.23: IMPACTO AL VARIAR EL NÚMERO DE REGISTRO POR HILO EN *REDUCE3* 59
FIGURA 3.24: IMPACTO AL VARIAR LA MEMORIA COMPARTIDA USADO POR BLOQUE EN *REDUCE3* 60
FIGURA 3.25: RESUMEN DE RECURSOS *REDUCE3* 61
FIGURA 3.26: IMPACTO AL VARIAR EL TAMAÑO DEL BLOQUE EN *RHO4* 62
FIGURA 3.27: IMPACTO AL VARIAR EL NÚMERO DE REGISTRO POR HILO EN *RHO4* 62
FIGURA 3.28: RESUMEN DE RECURSOS *RHO4* 63
FIGURA 4.1: IMAGEN PARA EL ESTUDIO DE LAS NORMALES 65

FIGURA 4.2: NORMALES OBTENIDAS CON RADIO 1 66
FIGURA 4.3: NORMALES OBTENIDAS CON RADIO 2 66
FIGURA 4.4: AMPLITUD DE LA IMAGEN 1 67
FIGURA 4.5: NORMALES DE LA IMAGEN 1 68
FIGURA 4.6: DIFERENCIAL DE ÁREA DE LA IMAGEN 1 68
FIGURA 4.7: *RHO* DE LA IMAGEN 1 69
FIGURA 4.8: AMPLITUD DE LA IMAGEN 2 69
FIGURA 4.9: NORMALES DE LA IMAGEN 2 70
FIGURA 4.10: NORMALES DE LA IMAGEN 2 70
FIGURA 4.11: DIFERENCIAL DE ÁREA DE LA IMAGEN 2 71
FIGURA 4.12: *RHO* DE LA IMAGEN 2 71
FIGURA 4.13: AMPLITUD DE LA IMAGEN 3 72
FIGURA 4.14: NORMALES DE LA IMAGEN 3 72
FIGURA 4.15: NORMALES DE LA IMAGEN 3 73
FIGURA 4.16: DIFERENCIA DE ÁREA DE LA IMAGEN 3 73
FIGURA 4.17: *RHO* DE LA IMAGEN 3 74
FIGURA 4.18: TIEMPO DE CÓMPUTO DE LA NORMAL CON RADIO DE 1 PÍXEL 75
FIGURA 4.19: TIEMPO DE CÓMPUTO DE LA NORMAL CON RADIO DE 2 PÍXEL 75
FIGURA 4.20: TIEMPO DE CÓMPUTO DEL DIFERENCIAL DE ÁREA 76
FIGURA 4.21: TIEMPO DE CÓMPUTO DE *RHO* 77
FIGURA 4.22: VALOR MEDIO Y DESVIACIÓN TÍPICA DEL TIEMPO DE CÓMPUTO DE CADA UNA DE LAS FUNCIONES 78
FIGURA 4.23: COMPARATIVA DE TIEMPOS 79

Índice de tablas

TABLA 3.1: RESUMEN DE TIEMPOS PARA EL CÁLCULO DE LA NORMAL DE TODOS LOS PÍXELES DE UNA IMAGEN 47
TABLA 3.2: DIFERENCIAL DE ÁREA PARA CADA PÍXEL 48
TABLA 3.3: RESUMEN DE TIEMPOS CÁLCULO DE ÁREA 51
TABLA 3.4: COMPARATIVA DE TIEMPOS DE *RHO* 52
TABLA 3.5: RESUMEN DE TIEMPOS DEL CÁLCULO DE *RHO* 63
TABLA 4.1: VARIACIONES TEMPORALES DE LA FUNCIÓN NORMAL 76
TABLA 4.2: VARIACIONES TEMPORALES DE LA FUNCIÓN DIFERENCIAL DE ÁREA 77
TABLA 4.3: VARIACIONES TEMPORALES DE LA FUNCIÓN *RHO* 77
TABLA 4.4: SUMATORIO DE TIEMPOS 78

Resumen

En el presente trabajo, se ha analizado un algoritmo para la corrección de un error de carácter no sistemático en cámara de tiempo de vuelo, determinando cuáles son las funciones con un mayor coste computacional. Estas funciones se han modificado para realizar su ejecución en GPU con un alto grado de paralelización, mediante su implementación en CUDA.

Las cámaras de tiempo de vuelo (*ToF*) proporcionan, además de la intensidad, información de profundidad en la escena, permitiendo obtener una imagen de distancias invariante a la iluminación. Estas medidas cuentan con diversos errores asociados, siendo uno de ellos el debido a la interferencia por multicamino (*MpI*).

Las interferencias por multicamino se deben a reflexiones múltiples dentro de la escena, es decir, la luz puede reflejarse en varias superficies antes de retornar al sensor, haciendo que la distancia medida pueda ser mayor a la real.

Además, se ha realizado un estudio del principio de funcionamiento de las cámaras *ToF*, analizando sus principales ventajas e inconvenientes, así como los errores que pueden presentar las medidas. A su vez, se ha realizado un estudio del modelo de programación en GPUs: manejo de memoria, paralelización de procesos, etc.

Para la validación de la implementación realizada se han ejecutado múltiples pruebas experimentales empleando imágenes reales grabadas con una cámara *ToF* de disponibilidad comercial. En cada prueba se ha medido el tiempo de cómputo, para posteriormente compararlo por el tiempo consumido por las mismas funciones implementadas en C y Matlab. En todos los casos, el tiempo de ejecución se ha reducido de forma notable, permitiendo validar el trabajo realizado.

Capítulo 1.Introducción

En los últimos años, se han desarrollado diferentes sensores que permiten obtener, además de la intensidad o color, una medida de profundidad (entendiendo como tal la distancia de cada punto de la escena al sensor). El uso de este tipo de sensores, se encuentra en auge en la actualidad, y tiene múltiples aplicaciones en diferentes ámbitos [1], tales como la robótica móvil [2], la recuperación de la estructura a partir del movimiento [3] o la detección y seguimiento de personas [4].

Estos sensores pueden dividirse en dos tipos en función de la tecnología utilizada para la medida de distancia. En el primer grupo se encuentran las cámaras que obtienen la distancia en función de las deformaciones que se producen en un patrón de luz estructurada emitido por la cámara [5], [6]. Estas cámaras proporcionan, además de la profundidad, una imagen en color de la escena, sin embargo, presentan algunas limitaciones, tales como la dependencia de la iluminación de la imagen, y la distancia mínima a partir de la cual se puede obtener una medida.

En el segundo grupo se encuentran las cámaras de tiempo de vuelo [7] en las que la distancia se obtiene de forma indirecta en función del tiempo de vuelo de una señal infrarroja modulada. Las cámaras *ToF* permiten obtener, además de una imagen de nivel de gris, una imagen de distancias invariante a la iluminación, siendo además la distancia mínima que puede medir bastante inferior al caso de las cámaras basadas en luz estructurada. Estas cámaras presentan varios tipos de errores, pudiéndose clasificar en errores de carácter sistemático y no sistemáticos. En el presente trabajo se aborda el análisis de un error de carácter no sistemático: la interferencia por multicamino.

Las interferencias por el multicamino se deben a reflexiones múltiples dentro de la escena, es decir, la luz puede alcanzar los objetos a lo largo de varios caminos, haciendo que la distancia medida pueda ser mayor que la distancia real. El error debido al multicamino, puede corregirse mediante la propuesta realizada en [22], que se explicará brevemente más adelante.

1.1. Objetivos

El objetivo principal de este trabajo es programar las funciones críticas del algoritmo para la corrección de la interferencia por multicamino (*MpI*) en cámaras de tiempo de vuelo (*ToF por sus siglas en inglés)* propuesto en [22], basándose en técnicas de computación en paralelo en GPU, para reducir los tiempos de cómputo, de cara a permitir el procesamiento de imágenes en tiempo real.

Para alcanzar el objetivo planteado, será necesario realizar un estudio exhaustivo del algoritmo propuesto, para detectar las funciones críticas. Dado que el algoritmo ha sido implementado previamente en C [8], y ha sido imposible alcanzar las especificaciones de tiempo real, se propone realizar el desarrollo basándose en técnicas de computación en paralelo en GPU (concretamente empleando lenguaje CUDA), para explotar los beneficios del procesamiento, de forma que sea posible mejorar estos tiempos.

Para la consecución del objetivo planteado, ha sido necesario llevar a cabo diferentes tareas, las cuales se describen brevemente a continuación.

Estudio del algoritmo de corrección Mpl en cámaras ToF

El primer paso para la realización del proyecto ha sido el estudio de la tecnología de funcionamiento de las cámaras *ToF*, así como de los diferentes errores existentes en la medida, prestando especial atención al efecto producido por el multicamino. También se ha estudiado con detenimiento el método empleado para la corrección del mismo.

El principal objetivo fue familiarizarse con el algoritmo realizado en [8] y [22] para la corrección de la interferencia por multicamino, así como con las diferentes funciones que lo componen.

Análisis de tiempos de cómputo en el algoritmo de partida

En la siguiente fase del trabajo se realizó un estudio más exhaustivo sobre el algoritmo de [8], [22] , distinguiendo posibles puntos críticos en su ejecución por incrementarse considerablemente los tiempos de cómputo.

Estudio de alternativas para la reducción del tiempo de procesamiento

El siguiente paso es el análisis de diferentes métodos para conseguir reducir notablemente los tiempos de ejecución en [8] y tratar de alcanzar los requisitos de tiempo real.

Se propone para ello, realizar un conjunto de funciones basándose en programación paralela en GPU's utilizando lenguaje CUDA.

Programación del algoritmo

Comprendido el método de corrección *Mpl*, se han implementado las funciones de mayor coste computacional de dicho algoritmo mediante programación paralela para mejorar los tiempos de ejecución.

Esta etapa ha sido evaluada empleando un conjunto de datos proveniente de capturas reales con cámaras *ToF*, con el objetivo de evaluar las prestaciones, puntos débiles y posibles mejoras de las mismas.

Realización de pruebas experimentales

Tras la finalización de la etapa de programación del algoritmo se realizaron nuevas pruebas experimentales, utilizando para ello distintas imágenes capturadas usando una cámara *ToF*, para validar el funcionamiento del mismo así como la mejora en los tiempos de ejecución, comprobando de esta forma la consecución del objetivo planteado.

1.2. Estructura de la memoria

Esta sección del capítulo describe cómo será la organización y la estructura de cada uno de los capítulos.

Para comenzar, en el Capítulo 2 se realiza un estudio de las principales características de la tecnología *ToF*, centrando el estudio en cámaras de profundidad. Del mismo modo se realiza una visión general de los errores que se producen en las cámaras *ToF*, centrándose en los debidos al multicamino. Además, se realizará una breve introducción a la programación en paralelo en GPU's basado en lenguaje CUDA.

A continuación, en el Capítulo 3 se explica el proceso de desarrollo de este trabajo, así como el análisis y diseño del sistema creado. Se irán mostrando resultados parciales de la programación, además de la exposición detallada de la implementación realizada para cada uno de los algoritmos.

El Capítulo 4 contiene los resultados experimentales obtenidos donde se muestra una comparación de los diferentes tiempos conseguidos para cada modelo de programación y los resultados de dichas funciones.

Para finalizar, el Capítulo 5 incluye las principales conclusiones del trabajo realizado, así como algunos posibles trabajos futuros.

Capítulo 2.Fundamento teórico

En este capítulo se exponen algunos conceptos teóricos necesarios para la comprensión del proyecto.

Para comenzar, en el apartado 2.1 se explica el principio de funcionamiento de las cámaras *ToF*, haciendo especial énfasis en la cámara SR-4000 de Mesa-Imaging, ya que es la utilizada en este trabajo. Se estudian, además, las principales ventajas e inconvenientes de la tecnología de las cámaras *ToF*, así como los errores debidos a esta tecnología.

A continuación, en el apartado 2.42.3 se expone brevemente el algoritmo de corrección del *MpI* cuyas funciones críticas se implementarán en GPU para la reducción del tiempo de procesamiento.

Finalmente, en el apartado 2.4 se realiza una descripción más detallada acerca de la programación en paralelo, centrándose en la programación en CUDA y sus principales ventajas, así como lo que implica en el trabajo.

2.1. Cámaras de profundidad basadas en tiempo de vuelo

En esta sección se analizan las características propias de la cámara utilizada para el desarrollo del proyecto. En concreto, se estudia la tecnología de las cámaras *ToF* (*Time of Flight* o Tiempo de Vuelo), haciendo especial énfasis en la cámara utilizada en este proyecto (SR-4000) que se muestra en la Figura 2.1.

Figura 2.1: Cámara *ToF* SR-4000

2.1.1. Principio de funcionamiento

En las cámaras *ToF*, la distancia de la cámara a cada objeto se obtiene de forma indirecta, en función del tiempo de vuelo de una señal infrarroja. Para ello, la cámara cuenta con un emisor infrarrojo que ilumina la escena con un haz de luz modulada, según la expresión (2.1), donde a_e es la amplitud de la señal emitida y $\omega = 2\pi f$, siendo f la frecuencia de la señal modulada.

$$r(t) = a_e \cdot e^{j\omega t} \tag{2.1}$$

Cuando la luz emitida alcanza un objeto, se refleja y puede detectarse en los píxeles de la cámara, siendo la señal detectada en el píxel *i*:

$$s_i(t) = a_d \cdot e^{j\omega(t+\Delta t)} = a_d \cdot e^{j\omega\varphi_d} \tag{2.2}$$

donde a_d es la amplitud de la señal recibida, y φ_d es la fase de la señal detectada.

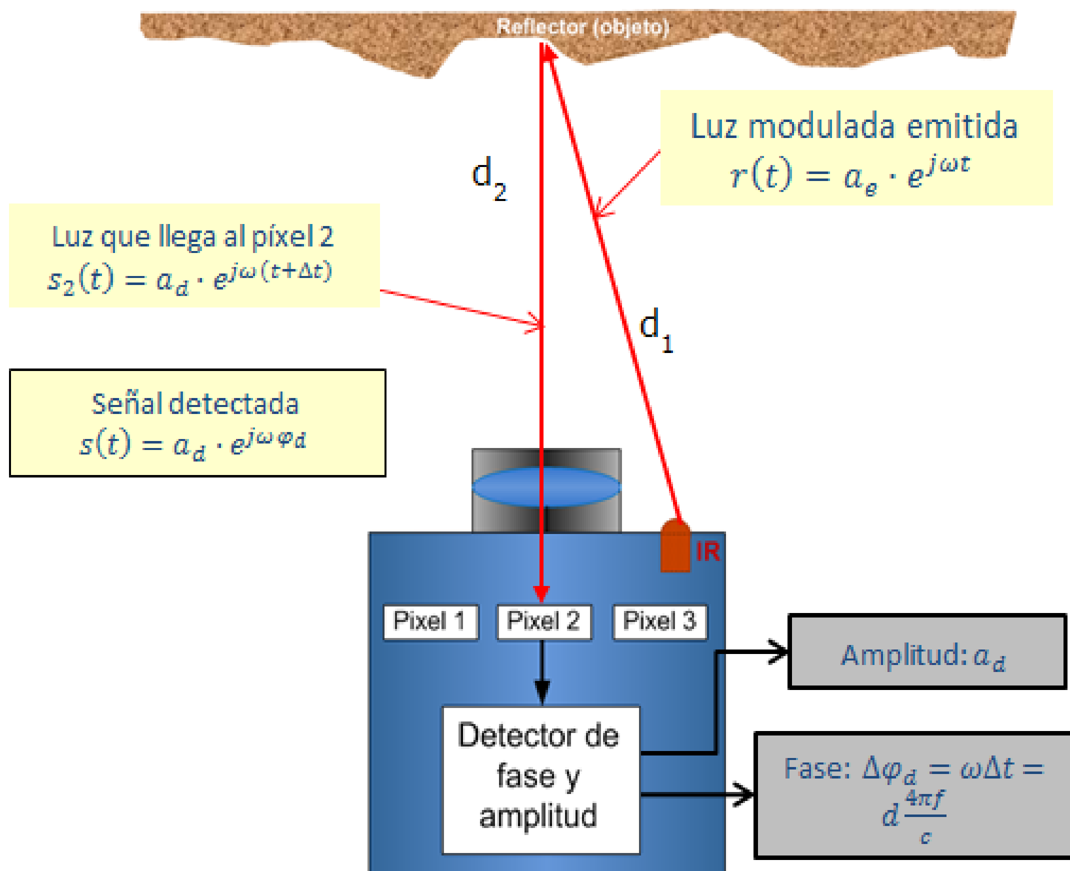

Figura 2.2: Principio de funcionamiento de una cámara *ToF*

En la Figura 2.2 se muestra un modelo simplificado del principio de funcionamiento de las cámaras *ToF*. Si suponemos que la distancia d_1 entre el emisor infrarrojo y el objeto es

aproximadamente igual a la distancia d_2 entre el objeto y la cámara, es posible obtener la distancia al objeto, a partir del tiempo de vuelo, usando la siguiente expresión.

$$\Delta t = \frac{d_1 + d_2}{c} \approx \frac{2d}{c} \Rightarrow d \approx \frac{\Delta t \cdot c}{2} \tag{2.3}$$

En las cámaras *ToF*, la medida del tiempo de vuelo se realiza de forma indirecta a partir de la diferencia de fase entre la señal emitida y la recibida. Dado que se conoce la frecuencia de modulación de la señal emitida, es sencillo recuperar el tiempo de vuelo, y estimar la distancia a partir de la diferencia de fase:

$$\Delta\varphi_d = \omega\Delta t \approx \omega\frac{2d}{c} = \frac{4\pi f}{c}d \tag{2.4}$$

A partir de la ecuación (2.5), se puede obtener la máxima distancia medible por una cámara *ToF* sin ambigüedad. Esto se produce cuando la diferencia de fase es de 2π radianes, ya que para diferencias mayores, se considerará: $\Delta\varphi_d = \Delta\varphi_d - 2\pi$. Así, la distancia máxima es función de la frecuencia de modulación de la cámara y la velocidad de la luz, según la siguiente expresión:

$$2\pi = \frac{4\pi f}{c}d_{max} \rightarrow d_{max} = \frac{2\pi c}{4\pi f} = \frac{c}{2f} \tag{2.5}$$

Cabe destacar, que tanto la amplitud como la diferencia de fase se obtienen a partir de la correlación cruzada entre la señal emitida (2.1) y la recibida (2.2). Además, para reducir el efecto del ruido y mejorar la calidad de las medidas de distancia, la correlación se realiza múltiples veces durante el tiempo de integración de la cámara. La medida de profundidad empleando tecnología *ToF* presenta diversas ventajas e inconvenientes, siendo las más significativas las que se presentan a continuación.

Ventajas de usar tecnología ToF

En comparación con sistemas binoculares o sistemas de triangulación, todo el sistema es muy compacto, la iluminación se coloca justo al lado de la lente.

Es muy fácil extraer la información de distancia de las señales de salida del sensor *ToF*, por lo tanto, esta tarea utiliza sólo una pequeña cantidad de potencia de procesamiento.

Las cámaras de tiempo de vuelo son capaces de medir las distancias dentro de una escena completa, pudiendo alcanzar tasas de hasta 100 imágenes por segundo, siendo idóneas para ser utilizadas en aplicaciones de tiempo real.

Además, la mayoría de las cámaras incorporan un sistema de eliminación de la iluminación de fondo, permitiendo reducir los efectos de cambios en esta iluminación, especialmente en entornos interiores.

Desventajas de usar tecnología ToF

Aunque la mayoría de la luz del fondo proveniente de la iluminación artificial o el sol se suprime, los pixeles de la cámara siguen generando respuesta dado su alto rango dinámico. La luz de fondo también genera electrones, que tienen que ser almacenados. El sol tiene una potencia de iluminación alrededor de 50 vatios por metro cuadrado, por lo tanto, si la escena de iluminación tiene un tamaño de 1 metro cuadrado, la luz del sol es 50 veces más fuerte que la señal modulada.

Si varias cámaras de tiempo de vuelo están adquiriendo imágenes al mismo tiempo, unas cámaras pueden alterar las medidas de las demás. Existen varias posibilidades para hacer frente a este problema:

- Multiplexación en el tiempo: un sistema de control que inicie la medida de las cámaras individuales consecutivamente, de modo que sólo una unidad de iluminación esta activa en cada instante.
- Diferentes frecuencias de modulación: las cámaras modulan su luz con diferentes frecuencias, dicha luz se recoge en los otros sistemas sólo como iluminación de fondo, pero no alteran la medida de la distancia.

2.1.2. Errores asociados a cámaras *ToF*

A pesar de ser muy adecuadas para un amplio rango de aplicaciones que requieren información 2.5D (como detección y localización de objetos), las medidas obtenidas pueden presentar errores, que es necesario tener en cuenta a la hora de trabajar con la información proporcionada.

En relación a estos errores, en [9] se realizan una serie de test que validan el correcto funcionamiento del modelo comercial de la empresa MESA-Imaging *(*SR-4000) y los errores que permanecen latentes. Se demuestra la eliminación de errores relativos a las auto-reflexiones producidas en el interior de la cámara, así como la dependencia de la orientación de la cámara respecto al objeto, al introducir un modo de auto ajuste del tiempo de integración. A su vez, se evidencia el error de medida cometido debido a la reflectividad de los objetos en la escena, provocando desviaciones inferiores de 5mm (asumible para muchas aplicaciones).

El rendimiento en la medida de distancia con cámaras *ToF* está limitado por una serie de errores, algunos de ellos son inherentes al principio de medida y otros dependientes de la escena monitorizada. A continuación se hará una clasificación de los errores refiriéndose a ellos como errores sistemáticos y errores no sistemáticos, respectivamente. En [10] se realiza un estudio de los diferentes errores presentes en las cámaras *ToF*. La investigación muestra un gran número de errores sistemáticos, la mayor parte de los cuales puede corregirse modelando el problema matemáticamente o empleando técnicas de calibración.

1. Errores Sistemáticos.

Se considera un error sistemático aquel que se produce de igual modo en todas las mediciones que se realizan de una magnitud. Puede estar originado en un defecto del instrumento, en una particularidad del operador o del proceso de medición. Estos errores se abordan desde una perspectiva de calibración, obteniéndose resultados que mejoran la precisión en un factor considerable. Entre los más relevantes, en el caso de las cámaras *ToF*, cabe citar los siguientes:

Error de amplitud. La mayoría de los errores de amplitud provienen del hecho de que la potencia de una onda decrece con el cuadrado de la distancia que cubre. Debido a esto la luz reflejada por los objetos decrece rápidamente con la distancia entre el objeto y la cámara. En [12] se desarrolla un sistema de corrección del error de amplitud haciendo uso de la información de distancia en cada píxel.

Error de distancia. La información de distancia en cámaras *ToF* no es siempre precisa. Los mayores errores se cometen con objetos lejanos de escasa reflectividad. Sin embargo, estos errores parecen ser de carácter sistemático y pueden ser compensados empleando la información que se dispone de antemano de las cámaras *ToF*. Dado que este tipo de error es más acusado en las zonas oscuras de la imagen, en [11] se propone identificarlos mediante un umbralizado de amplitud y se les aplica una corrección.

Patrón fijo de ruido. Consiste en la imagen media tomada sin aplicación de señal de entrada y es producida fundamentalmente por la corriente de oscuridad del elemento sensor, la cual es dependiente de la temperatura. Partiendo de que una imagen patrón de ruido se obtiene promediando varias imágenes en "oscuridad", por ejemplo, con el objetivo cubierto, [13] propone que la corrección de dicho ruido se realice simplemente restando la imagen patrón de ruido a la imagen objeto de ser corregida.

2. Errores No Sistemáticos o Aleatorios.

Se trata de errores que dependen directamente de la escena monitorizada y que por tanto no pueden corregirse con el conocimiento a priori de la cámara. Estos errores son de dificil modelado y por tanto limitan el rendimiento de las cámaras *ToF.* Entre ellos se destacan:

- Ruido de conversión fotónica, ruido de cuantificación y ruido de disparo o electrónico.
- Las reflexiones multicamino ocurren por oclusiones y objetos de carácter cóncavo (esquinas o huecos) de manera que la trayectoria realizada es mayor que la del camino directo. Este efecto degenera en que la superficie medida tome formas redondeadas.
- La dispersión de la luz se produce fundamentalmente en el interior de la cámara debido al sistema de lentes y el dispositivo óptico de manera que la luz proveniente de diferentes puntos se superpone a la medida original.

Reflexiones internas. Este fenómeno hace referencia a un tipo de error no local causado por reflexiones secundarias que tienen lugar entre la lente, el sensor y el filtro óptico.

- En [13] se realiza un modelado matemático del problema de las reflexiones internas mediante notación compleja. Se consiguen mejoras en la precisión de la medida de hasta un factor 10 dependiendo de la configuración de la escena.
- Por otro lado, en [14] realiza un repaso del efecto producido por las reflexiones internas ("*scattering artifacts*") en la cámara *ToF* comercial SR3000. Empleando una representación compleja se muestra la relación existente entre la señal óptica original y la influencia de dichas reflexiones. Se parte de la hipótesis de linealidad e invarianza espacial para crear un modelo donde las reflexiones internas puedan expresarse como una operación de convolución. El método de compensación de las reflexiones emplea una restricción del filtro inverso a *kernel* de convolución expresados como suma de *gausianas* cuyo funcionamiento sea posible en tiempo real. Los resultados obtenidos demuestran mejoras de entre el 95% y el 35% en el peor de los casos.

Efectos térmicos. El desplazamiento debido a la temperatura de ciertas partes de la cámara puede provocar errores en la medida de distancia.

Relación señal a ruido reducida. Este tipo de error varía según la cámara y el escenario empleado, por ejemplo, cámaras con diferente potencia de emisión. Además, la intensidad de la señal recibida depende de la distancia de los objetos en el escenario y la reflectividad de sus superficies. Iluminaciones de baja intensidad derivan en una mala relación señal a ruido que distorsiona la medida. Este problema puede ser resuelto fácilmente aumentando cuidadosamente la potencia de emisión o el tiempo de integración.

Artefactos de movimiento. Existen diversas técnicas para la adquisición de datos de profundidad empleando cámaras *ToF*. La mayoría realizan múltiples medidas secuenciales durante el tiempo de integración, que posteriormente emplean para obtener la distancia. Esto implica que si la cámara o cualquier elemento de la escena se desplazan durante la adquisición de los datos, un mismo píxel tomará información proveniente de distintas posiciones. El movimiento de la escena provoca un emborronamiento debido a que la información es falseada por varios píxeles vecinos en el sensor. En [15] se propone un método de corrección de los artefactos de movimiento basándose en la distinción entre los efectos del movimiento lateral y axial, y trata ambos efectos aplicando los resultados del flujo óptico a imágenes consecutivas de fase y de profundidad (10 fps empleando GPU).

Interferencia por multicamino. Se trata de un efecto inherente al principio de funcionamiento de las cámaras *ToF* que puede alterar la precisión de sus medidas en varios centímetros. La señal puede reflejarse en varias superficies antes de volver al elemento sensor, de manera que el trayecto recorrido sea superior al que realizaría en su camino directo. Dicho efecto se hace más acusado en las inmediaciones de esquinas o huecos, provocando que su apariencia se torne redondeada, tal como se muestra en la Figura 2.3.

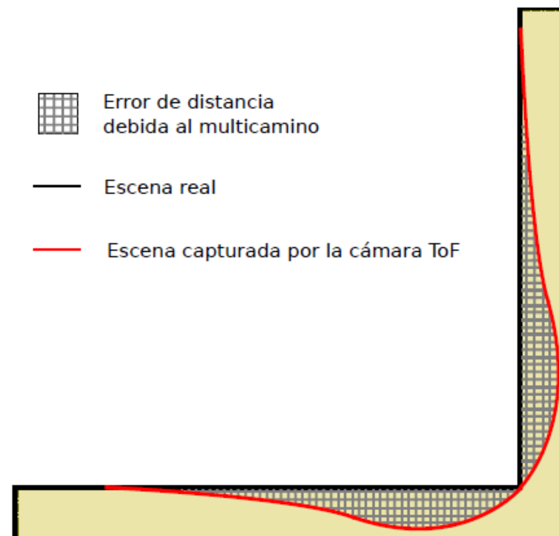

Figura 2.3: Efecto generado por la presencia de *Mpl*

En [16] se demuestra experimentalmente el impacto del multicamino empleando el ejemplo de una esquina. Debido a este efecto, la esquina se representa completamente imprecisa y los planos ortogonales obtenidos se abren hasta aproximadamente 120°.

Por otro lado, en [19] se demuestra el efecto del multicamino en un sistema de robótica. Una cámara *ToF*, sujeta a un robot, se emplea para estimar la dinámica del robot y acumular un mapa 3D. Los experimentos demuestran que el multicamino depende de la configuración y reflectividad de la escena. Eliminando los objetos reflectores o reduciendo su reflectividad la estimación de la dinámica del robot se ve mejorada en un 50%.

En [17], [18] y [13] se muestra la necesidad de eliminar las principales fuentes de error en las cámaras *ToF* para conseguir medidas de profundidad precisas. Entre ellas se destaca el error producido por las múltiples reflexiones en una escena y propone un método para su corrección basado en el uso de luz estructurada. Se ilumina la escena con la fuente habitual y a continuación se introduce luz estructurada. De esta manera, tomando como medida la información proporcionada por ambas fuentes, la señal debida al camino directo se verá reforzada en un factor mayor que la proveniente del multicamino, pudiéndose discriminar fácilmente.

En [20] se propone una aproximación para la compensación de dicho efecto donde se tiene en consideración la estructura de la escena medida. Se parte de la suposición de que todas las superficies se comportan como fuentes *lambertianas* y que la interferencia direccional se utiliza para propósitos de simulación. Con el objetivo de compensar la interferencia multicamino en un píxel concreto, se resta el error relativo de todas las contribuciones multicamino en la escena. Esto se realiza para todos los píxeles de la imagen empleando las medidas de distancia y amplitud. Debido a las exigencias de eficiencia, se realizan varias simplificaciones que alejan al resultado del valor óptimo teórico: las esquinas no son emisores *lambertianos* perfectos y las reflexiones producidas por las superficies que no están dentro del campo de visión no son consideradas, cuando en ocasiones tienen una aportación considerable.

2.2. Cámara SR-4000

En este trabajo se empleará la cámara SR-4000, cuyas características principales se presentan a continuación.

Externamente, la cámara utilizada se divide en dos partes principales como se muestra en la Figura 2.4:

- **Cubierta de Iluminación LED**: protege a los LED permitiendo al mismo tiempo transmitir la luz.
- **Filtro óptico:** permite que sólo la luz de longitudes de onda cerca de la de los LED de iluminación pueda pasar a la lente de la cámara, permitiendo la supresión de la iluminación de fondo.

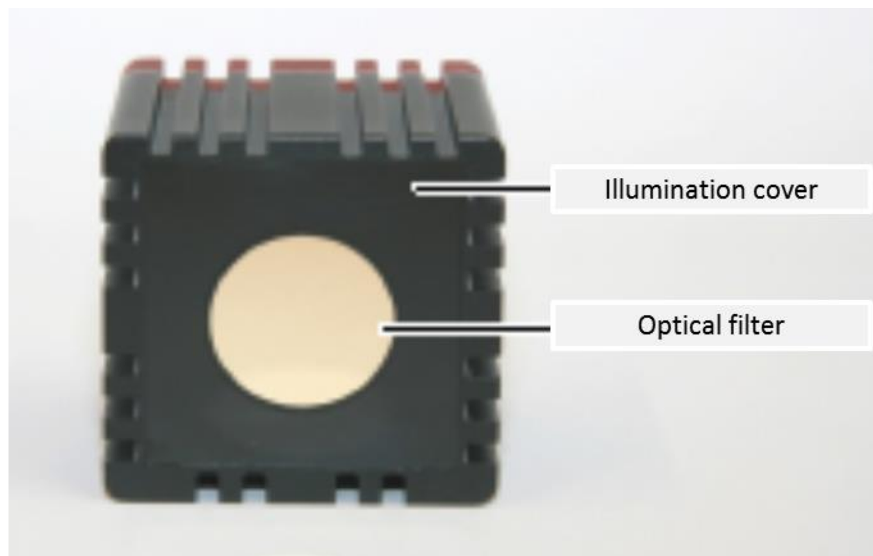

Figura 2.4. Regiones de la cámara

Con la cámara SR-4000 es posible capturar nubes de puntos con una distancia máxima de 5 o 10 metros, dependiendo de la frecuencia de modulación que emplee el modelo concreto utilizado, con una precisión en la medida de $\pm$15 milímetros, como se muestra en la Figura 2.5. En la misma figura también puede apreciarse que cuando el objeto supera el rango de distancia admisible de la cámara, hace imprecisa la obtención de la distancia real.

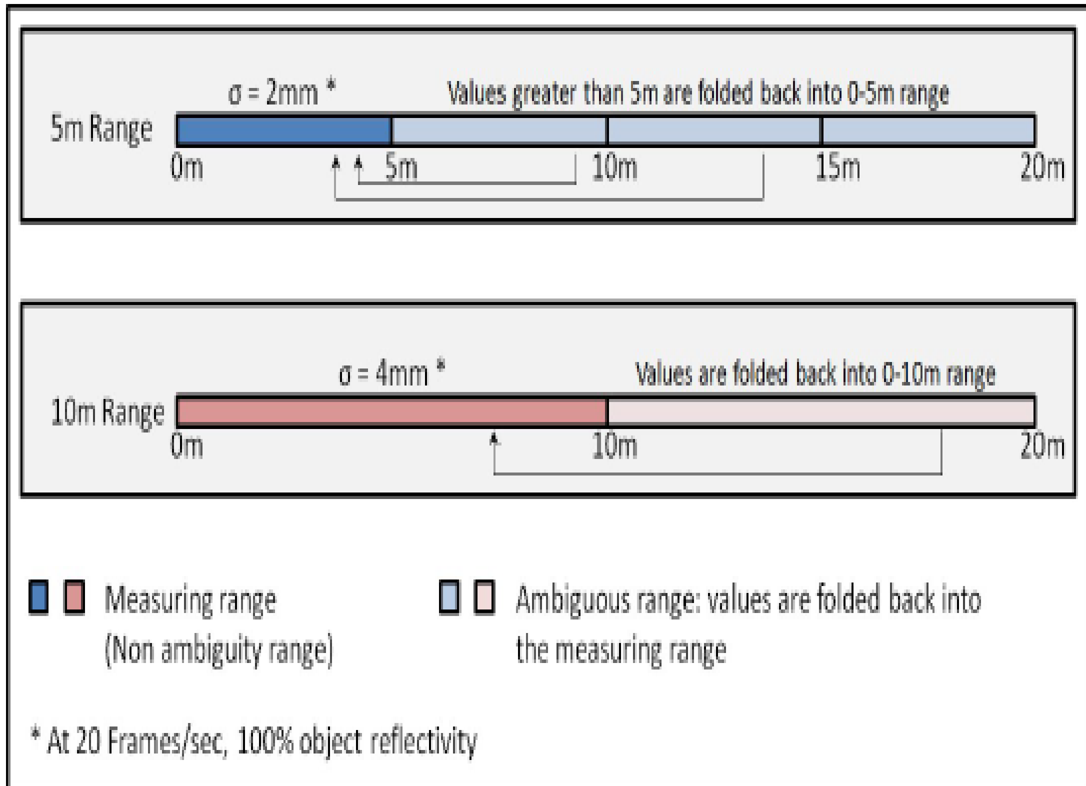

Figura 2.5: Eficiencia en la adquisición respecto a la distancia

Las medidas proporcionadas por la cámara SR-4000 están referidas a un sistema de coordenadas ubicado en la parte delantera de la carcasa, tal como se muestra en la Figura 2.6

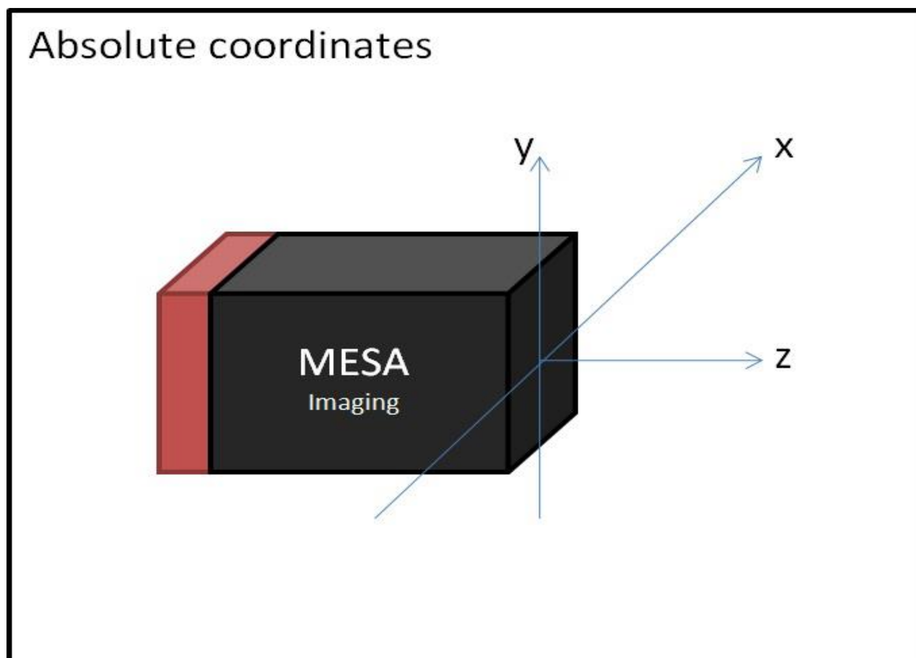

Figura 2.6: Sistema de coordenadas de la cámara

Datos de salida de la cámara

La cámara genera con cada adquisición varios archivos de salida, imágenes en falso color y ficheros con parámetros medidos. A continuación se explican los diferentes archivos de salida:

1- **Imagen de distancia**: contiene los diferentes valores de la distancia medida de cada píxel de la cámara al objeto encontrado enfrente.
2- **Imagen de intensidad en escala de grises**: con esta imagen se muestra la intensidad que tiene cada píxel. Empleando la ley del cuadrado de la distancia, en la que los objetos más alejados tienen menor iluminación, se deduce que la amplitud de la señal (intensidad) es mucho menor para los objetos a mayor distancia.
3- **Fichero de coordenadas XYZ**: incluye las coordenadas espaciales del objeto encontrado enfrente para cada píxel de la cámara, siendo Z la distancia al objeto.

Características destacables de la cámara SR4000

De acuerdo con el fabricante, la cámara SR4000 utilizada en este trabajo presenta las siguientes características:

- Medidas estables y continuas garantizadas mediante el diseño óptico de auto-calibración.
- Medidas independientes del color y reflectividad del objeto. Se consiguen medidas estables en repetitividad y precisión incluso en objetos de distintos colores y reflectividad dentro de la misma imagen.
- Se dispone de cámaras de dos rangos de medidas de 5m y 10m permitiendo más flexibilidad y precisión en la medida si se conoce la distancia a la que se medirán los objetos.
- Posibilidad de hacer medidas con multicámaras mediante la utilización de múltiples frecuencias de iluminación (hasta 3 cámaras).
- La supresión de luz de fondo en píxel permite medidas de mayor precisión, incluso en condiciones de luz de fondo desfavorable.
- El modo de adquisición seleccionable facilita las medidas en modo continuo o mediante *trigger*. El modo *trigger* es muy útil para sincronizar la cámara con otros dispositivos.

2.3. Corrección del MpI

Como ya se ha comentado, el *MpI* puede alterar la precisión de sus medidas en varios centímetros, debido a que la señal puede reflejarse en varias superficies antes de volver al elemento sensor, de manera que el trayecto recorrido es superior al que realizaría en su camino directo.

Para recuperar la posición real de cada punto de la escena hay que obtener el error de profundidad asociado a cada punto de la escena. Para ello se propone un modelo generativo del multicamino, ajustando iterativamente la escena a través de dicho modelo con la escena medida. Se aplica una corrección donde se tienen dos puntos aislados p_j y p_j cercanos a una esquina.

En la Figura 2.7 se muestra un ejemplo simplificado donde se tienen dos puntos aislados p_i y p_j cercanos a una esquina. La línea roja es la señal medida en el píxel *i* que se encuentra contaminada con *MpI* y la línea amarilla es la señal s_i contaminada de forma artificial por *MpI*. El parámetro que se necesita estimar para la corrección de cada punto en la escena es $\lambda_i = \frac{\Delta p_i}{\hat{p}_i}$, siendo $\hat{p}_i$ la posición medida del punto *i* respecto al sensor.

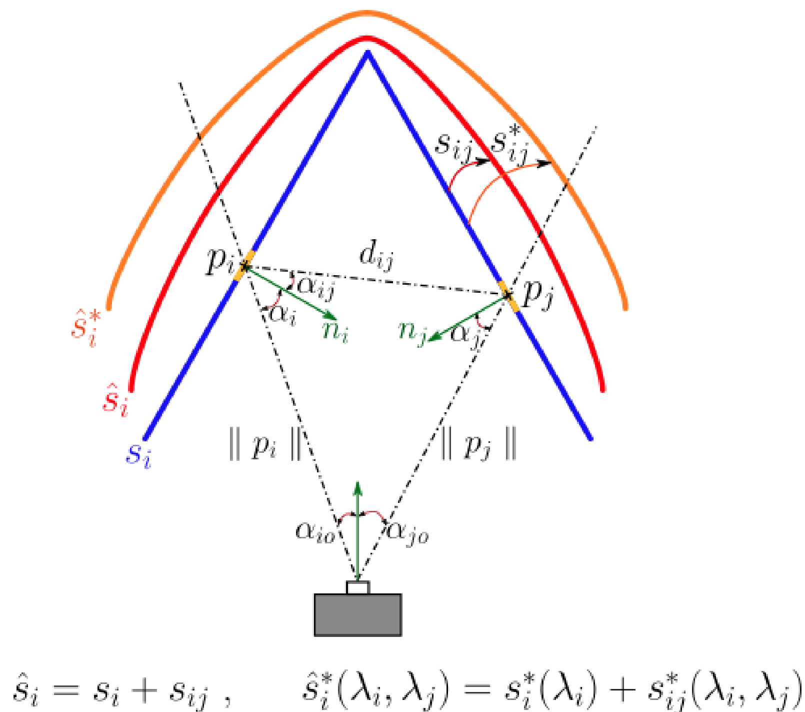

Figura 2.7: Diagrama del efecto del *MpI*

En este caso, si se minimiza el error de fase $\hat{s}_i$ *y* s_i^* se puede recuperar la escena corregida, mostrada en la Figura 2.7 con la línea azul. La demostración se presenta observando que la fase de s_i^*se iguala a la fase de $\hat{s}_i$ cuando los valores de λ_i, λ_j corrigen la escena.

Término debido al camino directo

Se obtiene la amplitud que llega al punto de la escena a_i^{col} y la amplitud irradiada a_i que retorna al píxel i, teniendo en cuenta:

- El *foreshortening* debido al ángulo $\propto_i$
- La señal emitida por cada píxel a_{out}
- La distancia entre el píxel y el punto

$$a_i^{col} = \frac{a_{out}}{\|p_i^2\|} \cdot \cos(\propto_i) \cdot dA_i \tag{2.6}$$

$$a_i = \frac{a_i^{col}}{\|p_i^2\|} \cdot \rho_i \cdot dA_o \cdot \cos(\propto_{io}) \tag{2.7}$$

donde,

- dA_o y $\propto_{io}$ corresponden al diferencial de área asociado a la superficie del píxel *i*
- ρ_i es el factor de albedo de la superficie en el punto, es decir, la energía absorbida por dicha superficie

Término debido al multicamino

Teniendo en cuenta que en este caso la energía impacta en el punto p_j irradiando al píxel *j*, además de impactar también en el píxel *i*, por lo tanto la distancia total recorrida por la señal se puede expresar como,

$$a_{ij} = a_{out} \frac{\rho_j \, dA_j \, \cos(\propto_j)}{\|p_j\|^2} \frac{\rho_i \, dA_i \, \cos(\propto_{ij})}{\|d_{ij}\|^2} \frac{dA_o \, \cos(\propto_{io})}{\|p_i\|^2} \tag{2.8}$$

Donde se ha incluido el término debido al multicamino debido al primer y segundo rebote de la señal con sus respectivos ángulos teniendo en cuenta las normales de cada superficie.

Al ser dA_o y $\propto_{io}$ comunes en el camino directo y en el multicamino, estos son absorbidos por la ganancia de la cámara.

Factor de albedo

El albedo es el porcentaje de radiación que cualquier superficie refleja respecto a la radiación que incide sobre la misma. Las superficies claras tienen valores de albedo superiores a las oscuras, y las brillantes más que las mates. Es una medida de la tendencia de una superficie a reflejar radiación incidente.

Éste es un factor importante a incluir ya que modela la cantidad de energía absorbida en cada punto de la escena por la cámara *ToF*.

Tenemos que ρ_i el factor de albedo para la superficie detectada en el píxel *i* del sensor. Para desacoplar acortamientos y decaimientos de amplitud se utiliza la medida de profundidad y las normales de cada píxel.

De esta manera se obtendrá el valor de la reflectividad en el píxel p_i^{rel} relativo al albedo máximo en la escena.

$$a_i^{origin} = \hat{a}_i \, \frac{\|p_i\|^4}{\rho_i \, dA_i \, \cos(\propto_i)} \tag{2.9}$$

$$\rho_i^{rel} = \frac{a_i^{origin}}{\max\{a_i^{origin}\}} \tag{2.10}$$

siendo a_i^{origin} la amplitud del píxel *i* en el mismo plano.

De esta forma se demuestra que, usando las medidas de la cámara *ToF*, se puede encontrar un mínimo de la función de coste (cf_i), con λ_i como solución que recupera la escena sin *MpI*. Para casos más generales la función de coste se expresa de la siguiente forma:

$$min\left(\sum_{i=1}^{N} \|cf_i\|_2^2\right) \tag{2.11}$$

donde,

$$cf_i = \Delta\varphi(\hat{s}_i) - \Delta\varphi(\hat{s}_i^*) = \Delta\varphi(\hat{s}_i) - \Delta\varphi\left(\hat{s}_i^*(\lambda_i) + \sum_j \hat{s}_{ij}^*(\lambda_i, \lambda_j)\right) \tag{2.12}$$

2.4. Arquitectura de cálculo paralelo en GPU

CUDA (*Compute Unified Device Architecture*) es un *framework* de desarrollo de aplicaciones paralelas de propósito general mediante el uso de dispositivos de aceleración gráfica (GPU's). Se trata de la alternativa desarrollada por NVIDIA en el campo de la GPU y está compuesta por un nuevo modelo de programación que permite a los desarrolladores el acceso a los dispositivos con capacidades CUDA, así como un conjunto de herramientas y librerías de desarrollo para facilitar la labor de los mismos en la programación y depuración de aplicaciones bajo este modelo.

En la Figura 2.8 podemos ver un esquema general del conjunto de elementos de la capa software de CUDA. Como vemos, la capa más baja comprende el propio motor de ejecución situado dentro de los dispositivos gráficos, seguido del soporte proporcionado por el *kernel* del sistema operativo. Sobre éste se sitúa por un lado el soporte *DirectX* para hacer uso directo de las llamadas del *kernel*, y por otro el driver CUDA que puede ser utilizado directamente o mediante una nueva capa de abstracción, como es el caso de *OpenCL*.

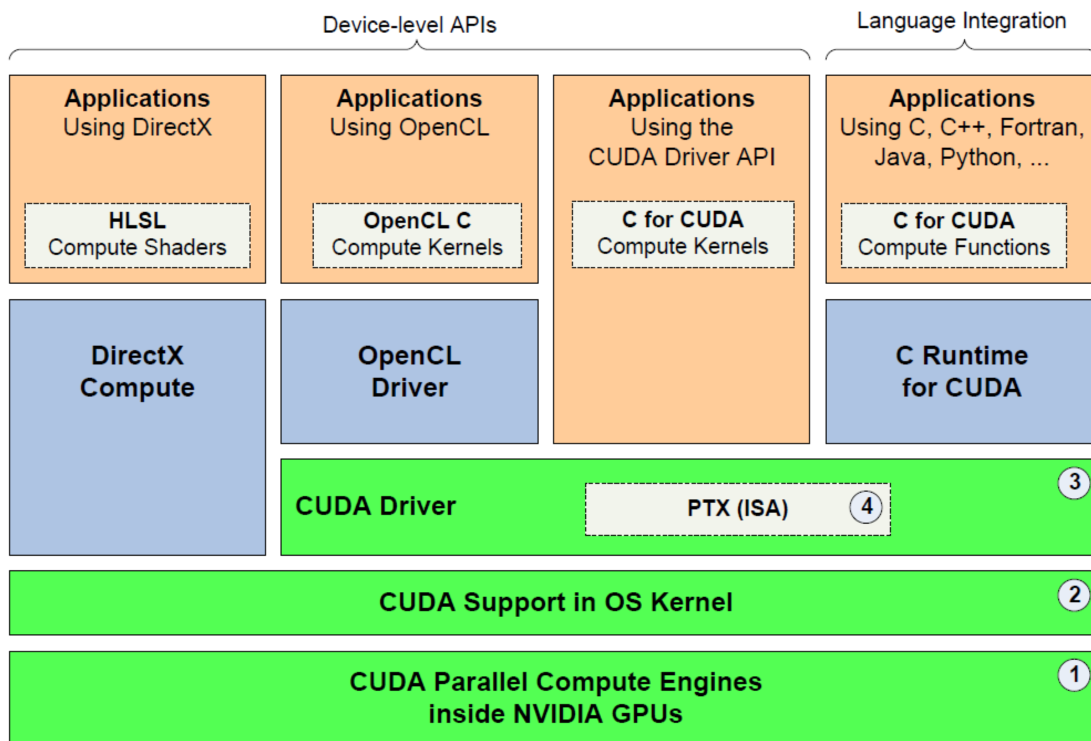

Figura 2.8: Elementos de la capa software en el modelo CUDA

2.4.1. Arquitectura general.

En la Figura 2.9 se muestra una visión esquemática de la arquitectura general de una GPU. Como vemos en ella, los dispositivos de procesamiento gráfico están formados por un conjunto de multiprocesadores (SM) de tipo MIMD (*Multiple Instruction stream*, *Multiple Data stream*), cada uno de los cuales a su vez contiene un grupo de procesadores (SP) de tipo SIMD (*Single Instruction stream, Multiple Data stream*). Se trata de procesadores de tipo vectorial, los cuales permiten ejecutar una misma operación sobre un conjunto o vector de datos simultáneamente.

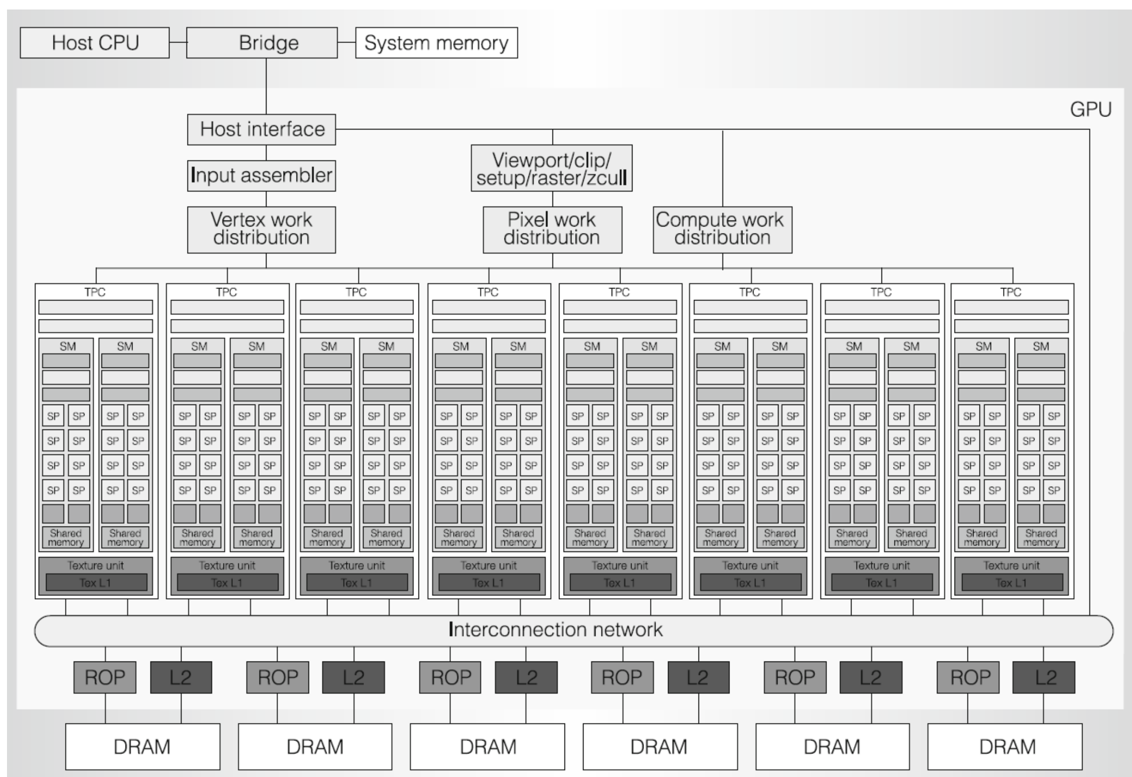

Figura 2.9: Arquitectura de una *GPU*

Sin embargo, la potencia de cada uno de dichos procesadores es inferior a la de una CPU en términos de frecuencia de operación o de operaciones soportadas, por lo que las mejoras en el rendimiento vienen del aprovechamiento del gran número de procesadores disponibles en la GPU, lo que nos permite lanzar y ejecutar un número elevado de hilos que ejecuten un conjunto de operaciones repetidas veces sobre diferentes datos.

Otro aspecto a tener en cuenta es que cada uno de los procesadores o *cores* realiza operaciones sobre datos de tipo entero o de coma flotante en simple precisión. La capacidad de operar sobre datos reales en doble precisión fue añadida a partir de la capacidad de cómputo 1.3 de CUDA, y dichas operaciones se realizan en unidades funcionales compartidas por todos los *cores* dentro de un mismo SM.

2.4.2. Modelo de programación.

Desde este punto de vista, la GPU se puede ver como un coprocesador de la CPU donde se ejecutarán las partes de código de una aplicación que sean intensivas en paralelismo de datos, en lo que puede considerarse un entorno de computación heterogénea. . Dado que no todo el código de una aplicación hará uso de este tipo de paralelismo, y los procesadores CUDA son inferiores en potencia a una CPU, se deberán utilizar ambas tecnologías para sacar el máximo partido al rendimiento de la aplicación. Así se diferencian dos partes en una aplicación que haga uso de CUDA: el código ejecutado en la CPU (*host*) y el ejecutado en la GPU (*device*).

Como dispositivo independiente de la jerarquía de memoria principal del sistema, la GPU necesitará de transferencias explícitas de datos entre ésta y su espacio de memoria, implicando esta acción una gran cantidad de tiempo adicional que afectará, en gran medida, al rendimiento de la aplicación.

Además de su correspondiente fase de ejecución, el *host* jugará un papel determinante en este entorno. Se verá implicado en la reserva y liberación de memoria en el dispositivo, así como en la realización de las transferencias de datos entre ambos ámbitos, y determinará cuándo el dispositivo comenzará a ejecutar cada uno de sus *kernel*. Se puede ver un esquema de las distintas fases en la Figura 2.10.

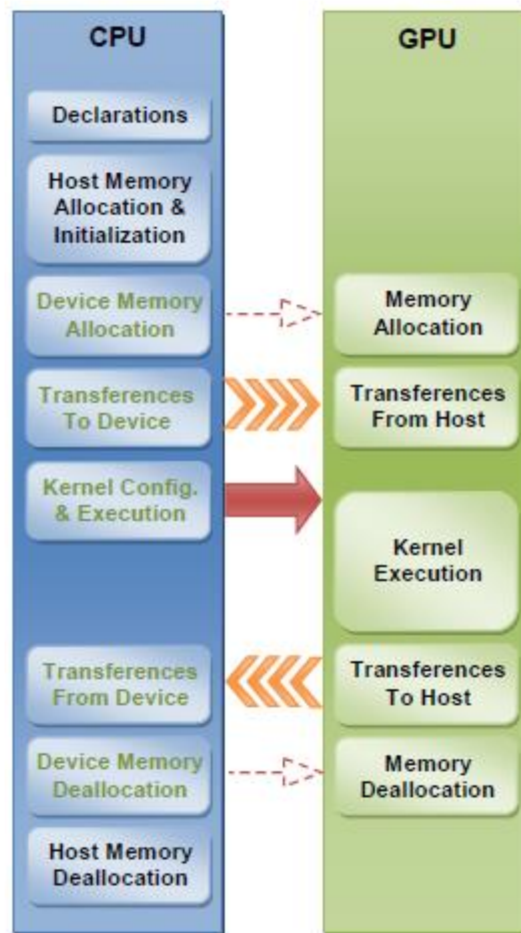

Figura 2.10: Fase de ejecución en CUDA

Entrando en un mayor nivel de detalle, la Figura 2.11 muestra un esquema del modelo de ejecución. Cada fase del código que está destinado a ejecutarse en una o varias GPUs recibe en nombre de *kernel*. Este código será ejecutado por todos y cada uno de los hilos del/los dispositivo/s, por lo que estará escrito en función del identificador de estos hilos, entre otros factores.

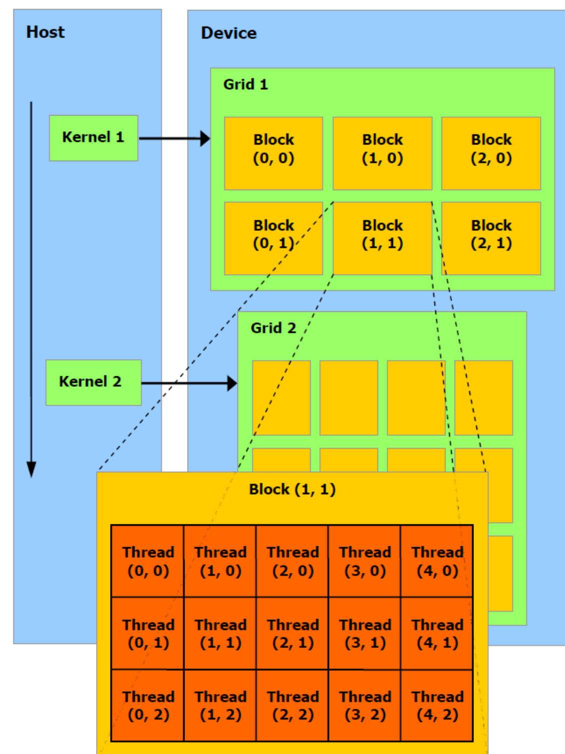

Figura 2.11: Modelo de ejecución en CUDA

Un bloque es una agrupación de hilos y viene a representar la unidad mínima de asignación entre dichos hilos y un multiprocesador del dispositivo. De esta forma, si el número de hilos por bloque es demasiado alto, tendremos menos bloques por multiprocesador ejecutándose en paralelo, y si es demasiado bajo, justo lo contrario. Existe un máximo de bloques e hilos que un multiprocesador puede manejar físicamente en paralelo. Para un aprovechamiento óptimo de los recursos, debemos tratar de que el número de hilos por bloque sea múltiplo del máximo de hilos que soporta cada multiprocesador, y sea lo suficientemente grande como para que los bloques puedan cubrir todos los recursos disponibles. Se recomienda que este valor coincida con una potencia de dos no excesivamente pequeña. También debemos intentar disponer de un número de bloques elevado para ocultar tiempos de latencia, y permitir que los multiprocesadores sigan trabajando cuando un bloque queda a la espera por algún motivo. A ser posible, esta cifra debe ser mayor que el doble del número de multiprocesadores del dispositivo.

Los bloques e hilos de cada *kernel* vienen englobados en una estructura denominada *Grid*. Esta estructura representa la materialización de cada *kernel* en un dispositivo, y es en la cual el programador establece el número de hilos por bloque y número total de bloques con los que se lanzará dicho código a la GPU. Así, se puede ejecutar el mismo *kernel* con la misma o diferente configuración de hilos y bloques sin que sea necesaria ningún tipo de modificación

de su código. Si bajamos un nivel más de abstracción, por debajo de los bloques nos encontramos a los *warps*, que no son más que una nueva agrupación de hilos que vienen a constituir la unidad básica de ejecución. Actualmente, los *warps* están formados por 32 hilos, lo que quiere decir que una misma instrucción será ejecutada a la vez por grupos de este tamaño. Cuando esto no sea posible porque distintos hilos dentro del mismo *warp* diverjan en instrucciones diferentes, cada hilo ejecutará su instrucción en serie con respecto a las demás distintas, disminuyendo, así, el paralelismo. También se encontrarán referencias a mitades de un *warp* o *half-warp*, ya que es la unidad de acceso a la memoria del dispositivo.

En definitiva, para aprovechar la potencia de estos dispositivos es necesario familizarizarse, al menos, con los conceptos de *kernel*, *Grid*, bloque e hilo, sin detenernos en demasiados detalles de bajo nivel. Desde este punto, se podrán desarrollar aplicaciones con un rendimiento más o menos aceptable, llevando a cabo una programación con granularidad de hilo.

2.4.3. Modelo de memoria.

La memoria es uno de los aspectos de las GPUs donde se podrán encontrar más lagunas de información, quizás, debido a temas de confidencialidad o a que los detalles existentes son de tan bajo nivel que resulta imposible su inclusión detallada en los manuales. Sin embargo, el correcto uso de los diferentes tipos de memoria que un dispositivo brinda será el mejor aliado si se quiere conseguir un mayor rendimiento.

En la Figura 2.8 se muestran los seis tipos de recursos que los dispositivos ofrecen para almacenar información, y el ámbito de visibilidad de los mismos: registros (*registers*), memoria local (*local memory*), memoria compartida (*shared memory*), memoria constante (*constant memory*), memoria de texturas (*texture memory*) y memoria global (*global memory*). Por defecto, la coherencia entre los recursos que requieren ser manipulados manualmente, en caso de ser necesaria, deberá ser mantenida por el programador. Este hecho debe ser aún más tenido en cuenta cuando nos referimos a la interacción entre la memoria del *host* y la memoria del dispositivo. No obstante, existen formas de que ambos compartan información de manera automática, como veremos más adelante.

Memoria global

La memoria global es la memoria más abundante, pero también la que se caracteriza por una mayor latencia y menor ancho de banda.

En CPU, sería equivalente a la memoria RAM. Se encuentra fuera del chip de procesamiento y es el medio principal para comunicar datos entre el *host* y el dispositivo. Esta comunicación, por defecto, debe ser manualmente especificada por el programador. El tiempo de vida de los datos es de todo el programa, y estos son visibles por cualquier hilo de cualquier bloque que se encuentre en ejecución en el dispositivo.

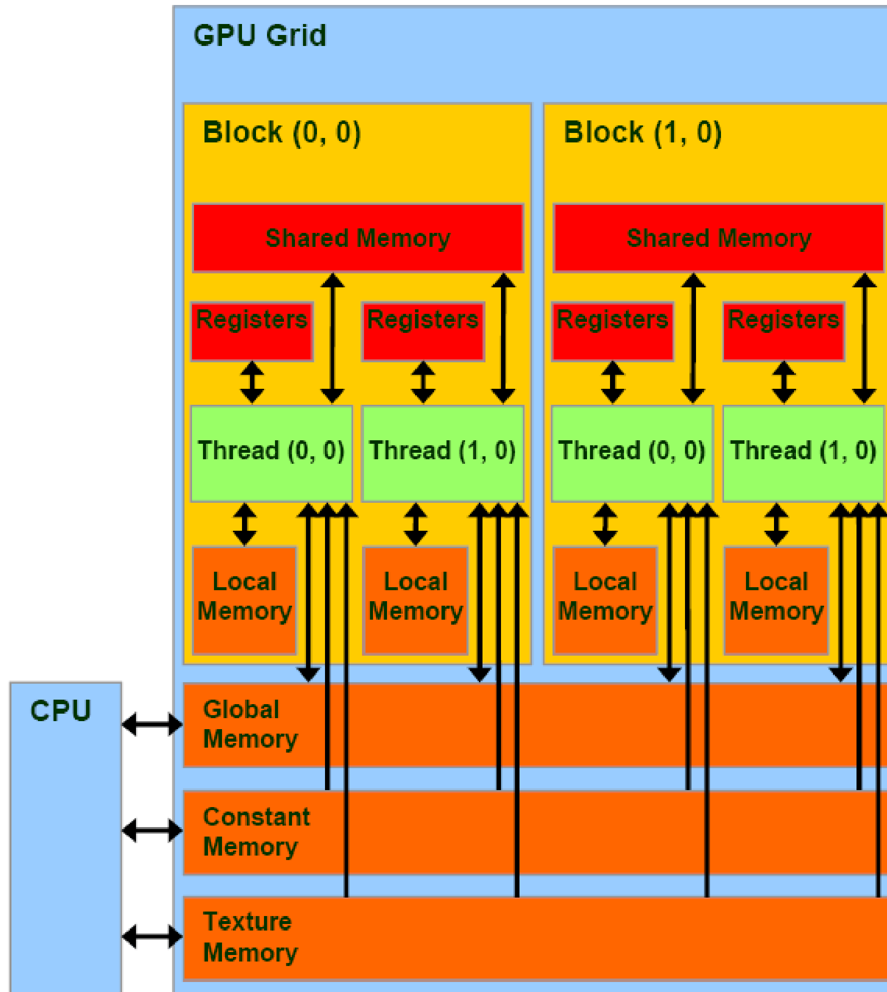

Figura 2.12: Modelo de memoria en CUDA

Los hilos acceden a ella en grupos de *half-warps*. Para obtener el mejor rendimiento, los accesos deben ser coalescentes. Este concepto ha ido evolucionando a la vez que lo han hecho las GPUs de propósito general, siendo cada vez menos restrictivo. Para dispositivos con capacidad de cómputo 1.0 y 1.1, un acceso será coalescente cuando en cada *half-warp*:

1- El tamaño de palabra accedido por hilo sea de 4, 8 o 16 bytes.
2- El primer elemento de cada *half-warp* debe estar alineado a un múltiplo de 16 veces su tamaño, para que todos los elementos se encuentren dentro del mismo segmento de memoria.
3- Los hilos deben acceder a los datos secuencialmente: el hilo *k* debe acceder a la palabra *k*. Pueden existir hilos que no soliciten acceso a ningún dato, siempre que esto no altere el orden para los hilos siguientes.

Si alguna de las restricciones no se cumple, la penalización será máxima, y se llevarán a cabo accesos secuenciales independientes de 32 bytes por cada hilo del *half-warp*.

Memoria local

La memoria local es una manera alternativa de utilizar memoria global, acotando la visibilidad y el tiempo de vida de los datos a nivel de hilo y de *kernel* respectivamente. No es otro tipo diferente de memoria y se puede asumir todo lo dicho anteriormente en lo que a características se refiere, con algunas salvedades. Normalmente, esta memoria es utilizada como área de desbordamiento de los registros o para almacenar estructuras de datos de varios elementos, locales a cada hilo. No es una decisión directa del programador el usarla o no, sino que es el compilador (estructuras de datos) y el *runtime* los que arbitrarán a su antojo su utilización. Esto supone que el hecho de realizar los accesos *coalescente* no esté en nuestra mano, por lo que serán estas entidades las encargadas de situar los datos en memoria global, de manera que se obtengan los mejores resultados de acceso.

Registros

Los registros son el recurso de almacenamiento más rápido de las GPUs, junto con la memoria compartida. Se encuentran dentro del chip de procesamiento y su visibilidad y tiempo de vida son similares a los de la memoria local. Como ya se ha comentado, al tratarse de un recurso limitado, cuando no quedan registros disponibles se utiliza la memoria local como área de desbordamiento. Al igual que ésta, el programador no puede controlar directamente su uso, aunque sí establecer algunas limitaciones.

Memoria compartida

La memoria compartida es uno de los recursos más valiosos del dispositivo. Se encuentra dentro del chip y, a diferencia de los registros y la memoria local, su visibilidad es a nivel de bloque. Esto quiere decir que todos los hilos del mismo bloque pueden compartir información utilizando este medio. Es un tipo de memoria bastante limitado, y será el programador el encargado de mover los datos entre esta y el resto de recursos manualmente. Además, su tiempo de vida es a nivel de *kernel* y su latencia puede equipararse a la de los registros, en el mejor de los casos. Sin embargo, esto no siempre será así. Los accesos a la memoria se llevan a cabo en grupos de *half-warps*, por lo que se dispone de 16 bancos con tamaño de palabra de 4 bytes. Cuando varios hilos, pertenecientes al mismo *half-warp*, intenten acceder al mismo banco, ya sea para alcanzar el mismo elemento o elementos diferentes, dichos accesos serán atendidos en serie, y por tanto, la latencia será mayor

Memoria constante

La memoria constante es un tipo de memoria dentro del dispositivo, cuya función es ofrecer un nivel de caché por encima de la memoria global. Como su nombre indica, se trata de una memoria de sólo lectura a nivel de GPU, por lo que los datos deberán ser transferidos/asociados a ella desde el *host*, antes de proceder con la ejecución de un *kernel* que acceda a esos elementos. Su visibilidad y tiempo de vida son similares a los de la memoria global, pero hay que destacar que no guarda coherencia con la misma.

Una lectura sobre la memoria constante resultará en un acceso a la memoria global del dispositivo en caso de fallo de caché, y un acceso a la memoria constante en caso de acierto.

Dicho acierto, en condiciones óptimas, será tan rápido como un acceso a un registro. Para que esto sea así, todos los hilos de un mismo *half-warp* deberán leer la misma posición de memoria. En caso contrario, los accesos a memoria serán serializados, y el coste del acceso se incrementará linealmente, en función del número de posiciones de memoria diferentes solicitadas dentro del mismo *half-warp*.

Actualmente, los dispositivos de NVIDIA cuentan con una memoria constante total de 64 KB. Este tamaño podría resultar interesante si se contempla desde el punto de vista de una memoria caché convencional de una CPU, donde es posible trabajar con un conjunto de datos de mayor tamaño al de su capacidad total, de manera transparente al usuario. Sin embargo, en este caso se encuentra la limitación de no poder utilizar ni un solo byte por encima de la capacidad de esta memoria, al menos de manera automática, por lo que el tamaño podría resultar algo escaso para objetivos de carácter general.

Memoria de texturas

La memoria de texturas es utilizada nativamente para leer texturas de imágenes en aplicaciones gráficas. Se puede decir que esta memoria pretende suplir las mismas funcionalidades que la memoria constante (nivel de caché sobre memoria global), pero con restricciones de acceso diferentes. Además, esta caché está organizada en dos niveles y diseñada para optimizar accesos con localidad de datos en dos dimensiones. Con ello, se consigue que los hilos de un mismo *half-warp* que lean datos próximos en alguna de las dimensiones obtengan un mejor rendimiento. Si los datos resultan no estar en esta caché, será necesario leer de memoria global. Esta memoria es menos restrictiva en cuanto a patrones de acceso, por lo que resulta de gran utilidad cuando nuestro programa no es capaz de adaptarse a las restricciones de la memoria global o constante, siempre que no se requieran escrituras.

2.4.4. Instrucciones comunes

*cudaMallocHost (void** ptr, size_t size)*

Función que asigna un tamaño en bytes de memoria *host* que es bloqueado y accesible al dispositivo. Con esta función, el driver realiza un seguimiento de los rangos asignados en la memoria virtual y automáticamente acelera las llamadas a funciones tales como *cudaMemcpy*()*. Desde que el dispositivo puede acceder directamente a la memoria, se puede leer o escribir con un alto ancho de banda que la memoria paginable obtenida con funciones como *malloc()*. La asignación de cantidades excesiva de memoria con *cudaMallocHost()* puede empeorar el rendimiento del sistema, ya que reduce la cantidad de memoria disponible para el sistema de paginación. Como resultado, esta función se utiliza mejor con moderación para asignar áreas de escala para el intercambio de datos entre el *host* y el dispositivo.

Parámetros:

- ptr – Puntero a memoria host asignada
- size - Solicita el tamaño de asignación en bytes

*cudaMalloc(void** ptr, size_t size)*

Esta función asigna en el dispositivo un tamaño en bytes de memoria lineal y devuelve un puntero **devPtr* a la memoria asignada, la cual es alineada de forma adecuada para cualquier tipo de variable.

Parámetros:

- devPtr – Puntero a la memoria asignada del dispositivo
- size – Solicita el tamaño de asignación en bytes

cudaMemcpy(void dst, const void* src, size_t count, enum cudaMemcpyKind kind)*

Copia un número de bytes desde el área de memoria apuntada por *src* al área de memoria apuntada por *dst*, donde *kind* puede ser *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost* o *cudaMemcpyDeviceToDevice*, especificando la dirección de la copia. Las áreas de memoria no deben solaparse.

Parámetros:

- dst – Dirección de memoria destino
- src – Dirección de memoria fuente
- count – Tamaño en bytes a copiar
- kind – Tipo de transferencia

*cudaFree(void**devPtr)*

Libera el espacio de memoria apuntada por devPtr, que debe haber sido devuelto por una llamada previa a *cudaMalloc()* o *cudaMallocPitch()*. Si *devPtr* es cero no se realiza ninguna operación.

Parámetros:

- devPtr – Puntero a la memoria para liberar del dispositivo

*cudaFreeHost(void**devPtr)*

Libera el espacio de memoria apuntada por *hostPtr*, que debe haber sido devuelto por una llamada previa a *cudaMallocHost()* o *cudaHostAlloc().*

Parámetros:

- ptr – Puntero a memoria para liberar

2.4.5. Estructura del programa

Volviendo a la Figura 2.11, se ve cómo es posible disponer de diferentes dimensiones a la hora de desplegar los hilos y los bloques dentro de un *Grid*. Dichas dimensiones permiten mapear los hilos a estructuras multidimensionales de una manera más sencilla. Se tendrán hasta tres dimensiones para indexar un *hilo* (variables threadIdx.[x,y,z]) y dos para los bloques (variables *blockIdx*. [x,y]). Además, *gridDim*.[x,y,z] y *blockDim*.[x,y,z] describen el tamaño de un *Grid* (en bloques) y un bloque (en *hilo*) respectivamente. Todas estas variables son preestablecidas y accesibles desde cualquier *kernel*.

Pasando a la práctica, la Figura 2.13 muestra cómo sería la declaración de un *kernel* que vendría a restar dos vectores y escribir la diferencia en un tercero. Cada *hilo* que ejecute dicho *kernel* tendrá un identificador global, `gid`, que le servirá para acceder a las posición de los vectores que le toque computar. Este identificador será calculado como el desplazamiento global del *hilo* con respecto al resto de bloques, más el identificador local del *hilo* dentro del bloque en el que se encuentra (`lid` o *threadIdx.x*). Este *kernel* es caracterizado como __global__, lo que significa que podrá ser invocado tanto desde el *host* como desde otro *kernel*.

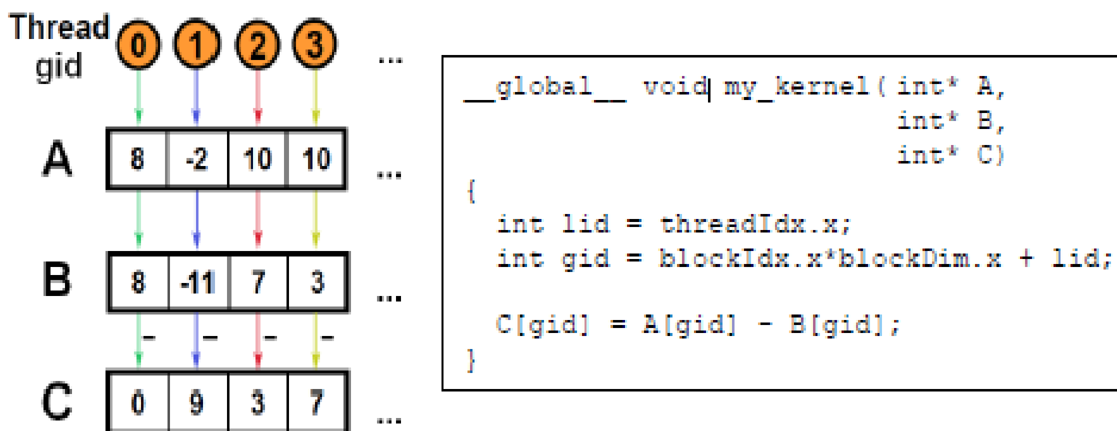

Figura 2.13: Ejemplo de *kernel*.

Suponiendo que el tamaño de los vectores es N (potencia de dos mayor o igual a 128), la Figura 2.7 describe dicha invocación con una configuración de 128 hilos por bloque y un *Grid* de (N/128) bloques. A, B y C representan a los vectores, donde los sufijos "_h" y "_d" indican si ese puntero hace referencia a la estructura que se encuentra en el *host* o en la memoria global del dispositivo, respectivamente.

```
//Declaration
int * A_h, *B_h, *C_h;  //Host pointers
int * A_d, *B_d, *C_d;  //Device pointers
//Host Memory Allocation
A_h = (int *) malloc (N * sizeof(int));
B_h = (int *) malloc (N * sizeof(int));
C_h = (int *) malloc (N * sizeof(int));

//Host structures initialization
...

//Device Memory Allocation
cudaMalloc((void **)&A_d, N * sizeof(int));
cudaMalloc((void **)&B_d, N * sizeof(int));
cudaMalloc((void **)&C_d, N * sizeof(int));

//Host To Device Transferences
cudaMemcpy(A_d, A_h, N*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B_h, N*sizeof(int),cudaMemcpyHostToDevice);

//Kernel Configuration
dim3 block(128);
dim3 grid(N/128);

//Kernel Launching (GPU Execution)
my_kernel<<<grid, block>>>(A_d, B_d, C_d);

//Device To Host Transferences
cudaMemcpy(C_h, C_d, N*sizeof(int),cudaMemcpyDeviceToHost);

//Host Memory Deallocation
free(A_h);
free(B_h);
free(C_h);
//Device Memory Deallocation
cudaFree(A_d);
cudaFree(B_d);
cudaFree(C_d);
```

Figura 2.14: Ejemplo de invocación de un *kernel*

La Figura 2.14 también deja constancia de las distintas fases que podemos encontrar en la invocación de un *kernel* habitualmente, aunque algunas de ellas podrían suprimirse o encontrarse en distinto orden, dependiendo de las necesidades del programa.

Se destaca que la invocación a un *kernel* es asíncrona, lo que quiere decir que el control de ejecución se devuelve al *host* cuando este ha terminado de lanzar el código a la GPU. Para conseguir que el host espere a que el *kernel* termine su ejecución, se puede hacer uso de la función *cudaThreadSynchronize(),* que actuará de barrera hasta que todos los hilos del *kernel* hayan finalizado su trabajo. El uso de esta función puede suprimirse si a continuación se va a realizar una transferencia de memoria, ya que la función *cudaMemcpy(),* además de ser bloqueante, espera a realizar su cometido hasta que el *kernel* no ha terminado de utilizar la GPU.

2.4.6. Maximización de la ocupación de los multiprocesadores

Ahora se centra el estudio en el grado de paralelismo. Para ello se va a estudiar cuántos de los recursos de los multiprocesadores están en uso. El concepto de ocupación refleja la cantidad de recursos en uso de un multiprocesador. Se define como la relación entre los *warps* activos y el máximo de *warps* activos de un multiprocesador. Entonces, a mayor ocupación mayor uso de los multiprocesadores. Por lo tanto, una buena estrategia de optimización debería ser tratar de elevar el ocupación al máximo.

Elevar la ocupación es sinónimo de elevar la capacidad de computación de los multiprocesadores, y es una buena estrategia para elevar el paralelismo. Teniendo en cuenta que los *warps* se componen de 32 hilos y que un multiprocesador puede ejecutar 1024 hilos, el máximo número de *warps* es 32. Sin embargo no siempre es posible conseguir ejecutar 32 *warps* a la vez. Los siguientes factores limitan el número de *warps* activos:

- El número de hilos por bloque y el número de bloques de hilos asignados al multiprocesador. Los multiprocesadores puede ejecutar a lo sumo ocho bloques de hilos en paralelo. Si el número de hilos por bloque es menor de 128 (4 *warps*) nunca se llegará a la máxima ocupación. Por otro lado, un bloque no se puede definir con más de 512 hilos (16 *warps*), por lo que al menos ha de haber dos bloques activos en el multiprocesador para obtener la máxima ocupación.

- El número de registros asignados a cada hilo. Cada multiprocesador cuenta con 16384 registros que asigna en bloques de 512 a los bloques de hilos. Esto hace que, habiendo hilos suficientes, cada hilo deba usar a lo sumo 16 registros para obtener la ocupación máximo.

- La cantidad de memoria compartida asignada a cada bloque de hilos. Cada multiprocesador cuenta con 16KB de memoria compartida. La cantidad de memoria compartida que usa cada bloque de hilos queda definida por el desarrollador a la hora de programar un *kernel*. Como ajustar el uso de la memoria compartida es otra estrategia de optimización.

La ocupación se puede calcular de forma estática según las siguientes ecuaciones. Los datos a conocer son el número de hilos por bloque (THREADS), número de registros por hilo (REGS) y cantidad de memoria compartida (SHMEM) por cada bloque de hilos.

$$Occupancy = \frac{warps_activos_por_SM}{warps_por_multiprocesador} \tag{2.13}$$

$$warps = \frac{\text{THREADS}}{threads_por_warp} \tag{2.14}$$

$$warps_activos_por_SM = bloques_activos_por_SM \cdot warps \tag{2.15}$$

$$bloques_activos_por_SM = \min(limite_por_warps, limite_por_reg, limite_por_shared) \tag{2.16}$$

$$reg_bloque = multiploSuperior(multiploSuperior(warps, 2) \cdot \text{REGS} \cdot threads_por_warp, 512) \tag{2.17}$$

$$limite_por_warps = \min\left(8, \frac{warps_por_multiprocesador}{warps}\right) \tag{2.18}$$

$$limite_por_reg = multiploInferior\left(\frac{total_registros}{reg_bloque}\right) \tag{2.19}$$

$$limite_por_shared = multiploInferior\left(\frac{total_shared}{shared_por_bloque}\right) \tag{2.20}$$

Como se verá en 2.4.6, no es necesario calcular la ocupación siguiendo estas ecuaciones, pero dan una idea de cómo se ocupan los recursos.

Por tanto, si el objetivo es maximizar la ocupación, hay que incrementar el número de *warps*, ya sea aumentando el número de hilos por bloque, disminuyendo el número de registros o disminuyendo la cantidad de memoria compartida. En el primer caso, se consigue aumentar la ocupación a base de ejecutar más hilos y menos bloques; y en los dos siguientes, a base de ejecutar más bloques en paralelo.

Únicamente queda discutir acerca del número de registros. En realidad, no existe un control directo sobre el uso de los registros de los multiprocesadores, al menos no a alto nivel. Todas las variables (excepto los *arrays*) definidas en un *kernel* se alojan en registros, sin embargo el cálculo de los registros totales que se necesitan los realiza el compilador aplicando sus propias estrategias de planificación de registros. Para saber cuántos registros utiliza un *kernel*, se debe compilar con la opción **–ptxas-options=-v**, o si se utiliza Visual Studio solo hay que visualizar el rendimiento del *kernel*. Al compilar con esta opción se indica en la salida de la compilación el número de registros que utiliza cada hilo del *kernel* compilado. Cualquier valor igual o menor de 16 es perfecto, ya que no impone ninguna restricción a la ocupación. Sin

embargo, valores más alto afectan a la ocupación en mayor o menor medida dependiendo del número de hilos por bloque. Como las ejecuciones se realizan por bloques de hilos, aunque todos los hilos menos uno tengan registros suficientes para ejecutarse el bloque entero debe esperar. Por ello, bloques grandes (de 256 o 512 hilos) que se vean limitados por el número de registros, merman la ocupación a razón de 25% y 50% respectivamente. Por otro lado, es posible forzar el número máximo de registros tratando de limitarlos en tiempo de compilación. La opción de compilación **–maxrregcount=X** limita el número de registros de los *kernel* compilados al valor X.

2.4.7. CUDA Occupancy Calculator

CUDA Occupancy Calculator es una hoja de Excel que ayuda a analizar la ocupación de multiprocesadores que nuestro *kernel*, lanzado al dispositivo, está realizando en función de los recursos que utiliza.

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	1,3

2.) Enter your resource usage:	
Threads Per Block	256
Registers Per Thread	20
Shared Memory Per Block (bytes)	8192

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	512
Active Warps per Multiprocessor	16
Active Thread Blocks per Multiprocessor	2
Occupancy of each Multiprocessor	50%

Physical Limits for GPU:	1,3
Threads / Warp	32
Warps / Multiprocessor	32
Threads / Multiprocessor	1024
Thread Blocks / Multiprocessor	8
Total # of 32-bit registers / Multiprocessor	16384
Register allocation unit size	512
Shared Memory / Multiprocessor (bytes)	16384
Warp allocation granularity (for register allocation)	2

Allocation Per Thread Block	
Warps	8
Registers	5120
Shared Memory	8192

These data are used in computing the occupancy data in blue

Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Multiprocessor	4
Limited by Registers / Multiprocessor	3
Limited by Shared Memory / Multiprocessor	2

Thread Block Limit Per Multiprocessor highlighted RED

Figura 2.15: CUDA Occupancy Calculator

La herramienta nos informa de qué recursos están limitando que no se utilice el resto de multiprocesadores que quedan disponibles. Así, se podrá valorar si se obtiene un mejor rendimiento reduciendo el uso de dicho recurso (en caso de ser posible) y aumentando la ocupación de multiprocesadores, o por el contrario, el uso actual del recurso proporciona mejor rendimiento, a pesar de que no se utilicen todas las unidades de cómputo disponibles.

Como se puede ver en el ejemplo de la Figura 2.15, la herramienta solamente necesita disponer del número de *hilos* por bloque de nuestro *kernel* (256), del número de registros por *hilo* (20) y del tamaño usado de memoria compartida por bloque (8192). Estos datos pueden obtenerse directamente con *Visual Studio*, entre otras opciones. Como resultado, la hoja de cálculo nos muestra la ocupación que estamos haciendo de la tarjeta (50%), y el recurso que está limitando ese valor (la memoria compartida).

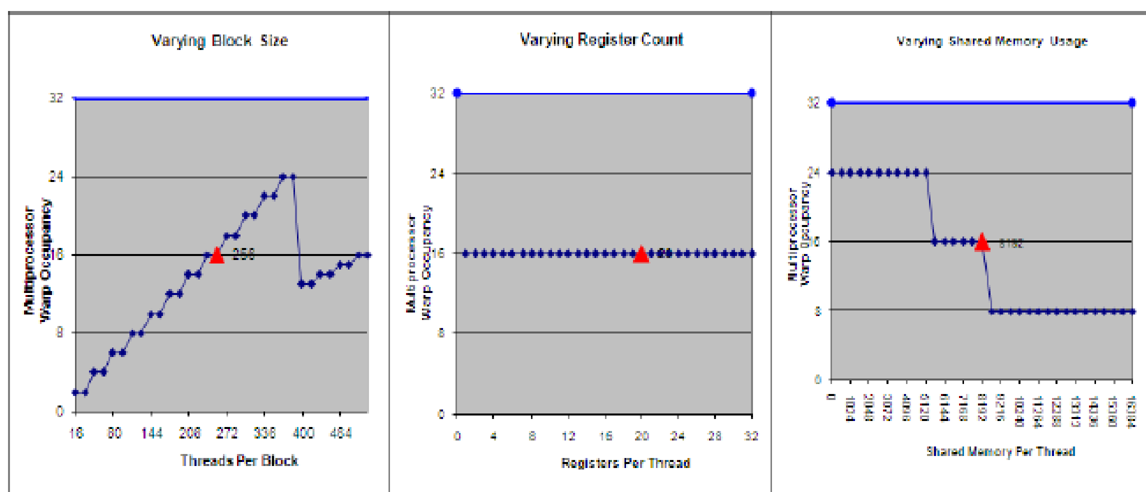

Figura 2.16: Gráficas de predicción de la ocupación

Las gráficas generadas por la utilidad (Figura 3.3) resultan muy interesantes para saber cómo va a evolucionar la ocupación de la GPU, en función de los tres parámetros que afectan a su valor. De esta manera, podremos planificar las modificaciones antes de llevarlas a cabo. En el ejemplo, según se puede observar, si reducimos el uso de memoria compartida permite alcanzar hasta un 75% de la ocupación. También se puede conseguir el mismo objetivo si aumentamos el número de *hilos* por bloque hasta 384.

2.4.8. Ventajas.

C para la GPU: Lenguaje de programación de alto nivel basado en estándares abiertos y ampliamente utilizado que aprovecha el poder de las GPU programables para dar paso a una nueva clase de aplicaciones y nuevos niveles de rendimiento en las operaciones de cálculo.

La capacidad de una supercomputadora: Picos de rendimiento de más de 500 *gigaflops* (miles de millones de operaciones en coma flotante por segundo) en aplicaciones de alta carga computacional.

Cálculo en múltiples GPU: Una sola CPU puede controlar varias GPU a través del *driver* de GPU *Computing* de CUDA, lo que da como resultado un rendimiento excepcional en las operaciones de cálculo. La capacidad de la GPU para resolver problemas a gran escala puede multiplicarse dividiendo el problema entre varias GPU.

Controladores de GPU Computing de NVIDIA: Se encargan de manejar los recursos de las GPU y una amplia librería de tiempo de ejecución para mejorar la gestión de los datos y la ejecución de los programas. Ofrecen un canal de transferencia de datos a alta velocidad y un *driver* mejorado que facilita las operaciones de cálculo con independencia del controlador de gráficos.

Arquitectura de cálculo con extraordinaria capacidad de procesamiento multihilo: Ejecuta miles de subprocesos (hilos) de forma simultánea para procesar en paralelo problemas matemáticos de alta carga computacional a gran velocidad. Su diseño modular permite utilizar varias GPU NVIDIA para ampliar la capacidad.

Caché de datos paralelos: Multiplica el ancho de banda efectivo y reduce las latencias al permitir que grupos de procesadores colaboren en el uso de la información contenida en la caché local. Los datos se copian menos veces y están disponibles al instante para todos los procesadores que compartan la misma caché de datos paralelos (*Parallel Data Cache*).

Transferencia de datos PCI Express a alta velocidad: Al disfrutar de menor latencia y un elevado ancho de banda, las aplicaciones de cálculo pueden aprovechar las tasas de transferencia de datos más altas que pone a su disposición la arquitectura PCI *Express* estándar.

Compatible con arquitecturas estándar: Tesla se basa en una arquitectura estándar y, por tanto, es compatible con los microprocesadores x86 de 32 y 64 bits de Intel y AMD, y con sistemas operativos de Microsoft y Linux.

Capítulo 3. Algoritmo de corrección del MpI

3.1. Introducción

En este capítulo se explican cada una de las funciones programadas basándose en técnicas de computación en paralelo en GPU, empleando lenguaje CUDA, y que permitan conseguir el objetivo de reducir los tiempos de cómputo del algoritmo previo, programado en C.

3.2. Diagrama de la propuesta de corrección MpI

La Figura 3.1 muestra el diagrama de la propuesta de compensación del *MpI*. La amplitud y la profundidad de las medidas realizadas por la cámara *ToF* (asumiendo dichas medidas contaminadas por *MpI*), se usan junto al modelo radiométrico para recuperar la escena original. Se aplica un proceso iterativo que busca la aportación del multicamino en la escena capturada por la cámara *ToF*.

En la Figura 3.1, las medidas de profundidad (caja verde) se usan para inicializar la solución (k=0); llamamos "solución (k)" a la escena de entrada corregida con el aporte del *MpI* para cada iteración k del lazo (caja azul). A su vez, las medidas de amplitud (escena en escala de grises) se utilizan para generar la contribución del multicamino en la solución, basada en el modelo radiométrico. El lazo obtiene el *MpI* y la solución, que añadidos, mejor se ajusten a la escena medida. Cuando el error medido entre la escena y la solución contaminada con *MpI* es menor que un umbral, se considera la solución de la escena corregida.

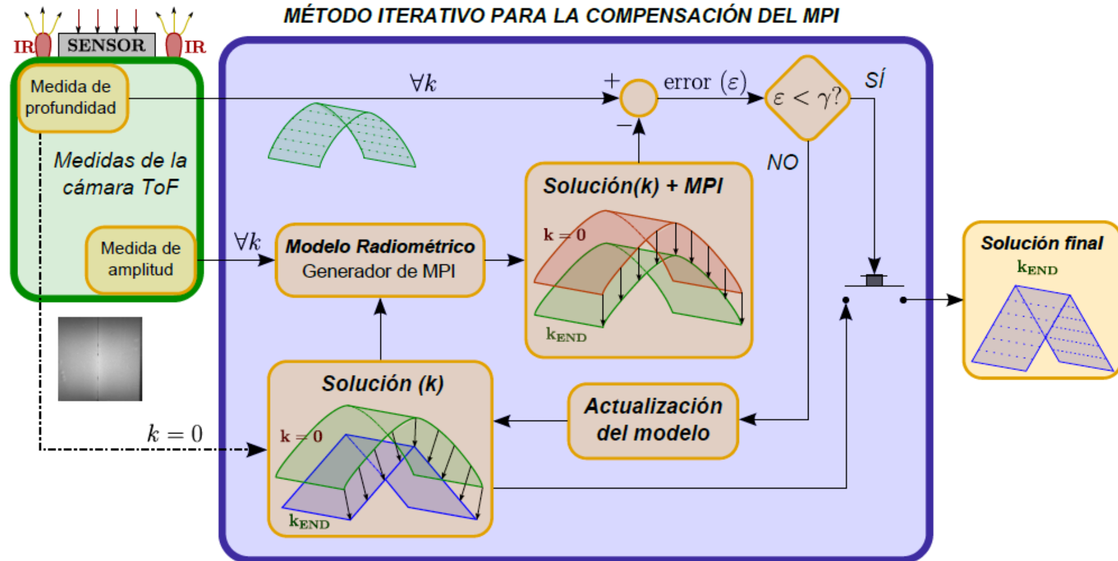

Figura 3.1: Diagrama general del algoritmo iterativo para la corrección *MpI*

Una vez explicado el proceso de corrección del *MpI* se procede al estudio del algoritmo de corrección implementado de forma que se relacionen las funciones con lo anteriormente expuesto. De esta forma se obtienen las funciones de mayor coste computacional en C, las cuales serán implementadas en CUDA.

Las funciones de mayor coste computacional son el cálculo de las normales de cada píxel de la imagen, la obtención de los diferenciales de área y el cálculo de los coeficientes relativos de absorción de cada punto de la escena. Otra de las funciones con mayor coste computacional es la minimización por Levenberg-Marquardt, pero no se contempla en el presente trabajo debido a que se considera que, dada la naturaleza secuencial de la minimización, su tiempo de ejecución no puede reducirse de forma significativa mediante la ejecución en GPU.

3.3. Condiciones iniciales

Los parámetros de entrada a las funciones programadas son las coordenadas 3D y la amplitud de cada píxel. Desde el *host* se leen ambos ficheros de datos y se copian en memoria.

En este punto cabe recordar que el *device* no puede leer datos del *host* y viceversa, por lo que es necesario reservar memoria de igual forma en el *device* y copiar los datos en él para trabajar en CUDA. Se comienza copiando ambas matrices de datos de CPU a GPU, pudiendo utilizarlas en los posteriores *kernel*.

De igual forma es necesario reservar memoria tanto en *host* como en *device* para cada uno de los datos de salida. Posteriormente los datos obtenidos en los *kernel* deben volver a copiarse al host para su posterior visualización.

3.4. Cálculo de las normales

La función estima el vector normal de un píxel dado aplicando minimización por mínimos cuadrados. Primero, se estiman los parámetros del plano usando un núcleo de vecindad y a continuación se calcula el vector normal.

Además se tiene que dar que las normales apunten hacia el detector, es decir que la componente Z del vector normal sea definida negativa.

Para obtener las normales de cada uno de los píxeles se comienza obteniendo un núcleo de vecindad próximo al píxel en cuestión y posteriormente se aplica mínimos cuadrados para hallar la ecuación del plano de este conjunto.

3.4.1. Obtener núcleo de vecindad

Dado un píxel de la imagen se obtiene su núcleo de vecindad calculando la distancia a los píxeles vecinos. Como píxeles vecinos se definen los que se encuentran en un radio de vecindad de uno o dos píxeles, pudiendo configurarse este parámetro para obtener mayor precisión.

Este método agiliza en gran medida el tiempo de cómputo, siendo sólo necesario calcular un número reducido de distancias entre el píxel bajo estudio y el resto de sus vecinos.

Entre todos estos vecinos propuestos sólo se consideran vecinos reales aquellos que se encuentran a una distancia inferior a 15 *cm* respecto al píxel central, descartando así posibles puntos de ruido o los muy alejados.

Resumen de recursos

La Figura 3.2 muestra el número de warps que se dispondría si se variase los hilos por bloque, consiguiendo para este caso con 512 hilos, en un total de 50 bloques en el *Grid*.

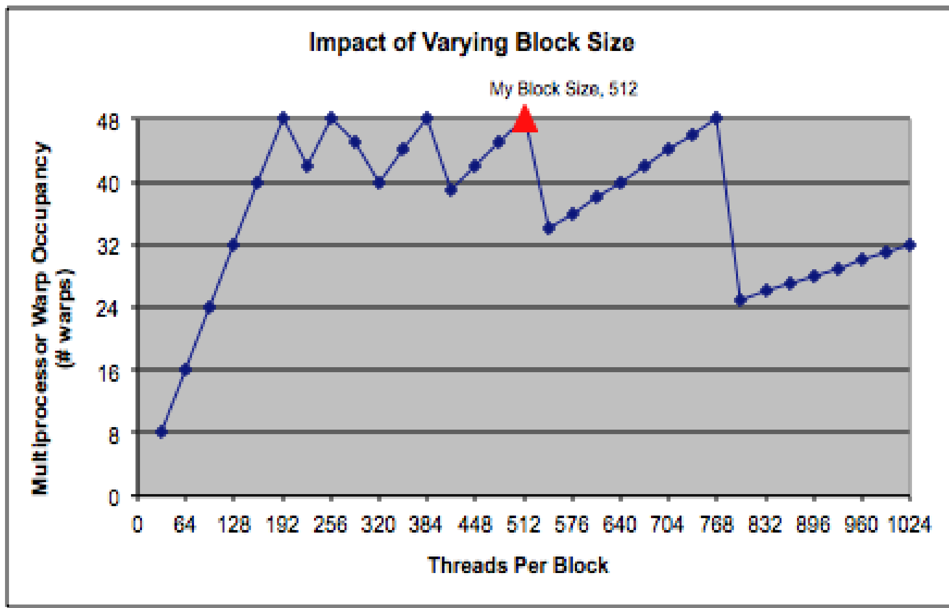

Figura 3.2: Impacto al variar el tamaño del bloque en *kernel1*

Este *kernel* emplea 12 registros por hilo, mostrando en la Figura 3.3 el número de *warps* máximo que es posible conseguir.

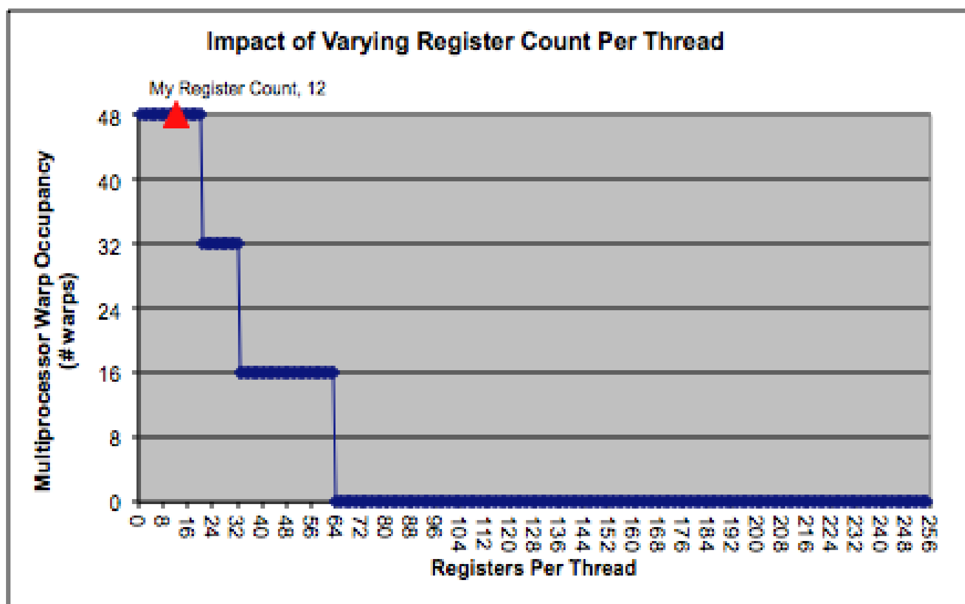

Figura 3.3: Impacto al variar el número de registro por hilo en *kernel1*

Sustituyendo en las ecuaciones descritas en el apartado 2.4.6

$$warps = \frac{\text{THREADS}}{threads_por_warp} = \frac{512}{32} = 16 \tag{3.1}$$

$$limite_por_warps = \min\left(8, \frac{48}{16}\right) = 3 \tag{3.2}$$

$$reg_bloque = multiploSuperior(multiploSuperior(16,2) \cdot 12 \cdot 32, 512) = \\ = 6144 \tag{3.3}$$

$$limite_por_reg = multiploInferior\left(\frac{32768}{6144}\right) = 5 \tag{3.4}$$

$$bloques_activos_por_SM = \min(3, 5) = 3 \tag{3.5}$$

$$warps_activos_por_SM = 3 \cdot 16 = 48 \tag{3.6}$$

$$Occupancy = \frac{48}{48} = 1 \tag{3.7}$$

El mismo resultado que las ecuaciones se obtiene con CUDA Occupancy Calculator como se muestra en la Figura 3.4.

2.) Enter your resource usage:	
Threads Per Block	512
Registers Per Thread	12
Shared Memory Per Block (bytes)	0

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	**1536**
Active Warps per Multiprocessor	**48**
Active Thread Blocks per Multiprocessor	**3**
Occupancy of each Multiprocessor	**100%**

Figura 3.4: Resumen de recursos *kernel1*

3.4.2. Obtener ecuación de plano

El método aplicado para obtener el plano es minimización por mínimos cuadrados. Sería conveniente aplicar también ajuste de paraboloides y comparar ambas desviaciones cuadráticas, quedándose con la menor de ellas, pero para reducir el consumo de registros y minimizar el tiempo de cómputo sólo se aplica mínimos cuadrados.

En la Figura 3.5 se muestra la forma de obtener un plano mediante mínimos cuadrados para que su desviación cuadrática sea mínima.

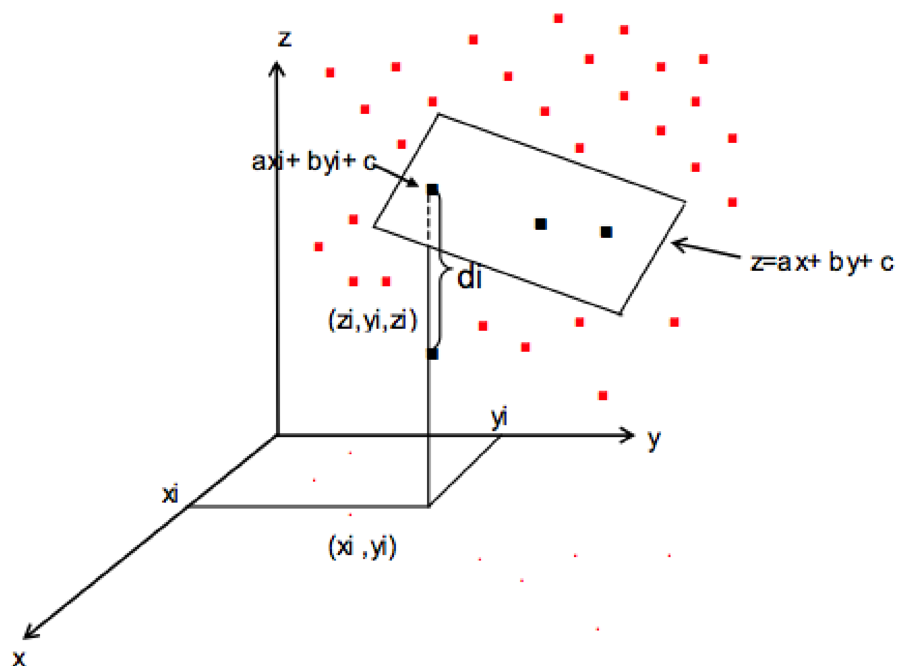

Figura 3.5: Obtención de un plano

Para poder aplicar este método y conseguir un plano es necesarios tener al menos 3 puntos en el núcleo de vecindad, en el caso de no tener al menos 3 puntos no es posible obtener la ecuación del plano para ese determinado píxel.

Para un plano que se ajuste aproximadamente a todos los puntos observados, algunas de las desviaciones serán positivas y otras negativas, pero sus cuadrados serán todos positivos, de donde en la expresión $\sum_{i=1}^{n} d_i^2$ se considera como equivalentes una desviación positiva *d*, y otra negativa –*d*. Esta suma de cuadrados de las desviaciones depende de la elección de a, b y c. Es siempre positiva y sólo es igual a cero en el caso de que todas las desviaciones sean nulas; es decir, cuando a, b y c se eligen de modo que el ajuste es perfecto.

Sea o no posible hallar un plano que proporcione un ajuste perfecto, el método de los mínimos cuadrados dice: tómese como plano $z = ax + by + c$ de ajuste óptimo, aquel para el cual la suma de los cuadrados de las desviaciones, $d_1^2 + d_2^2 + \ldots + d_n^2$ es mínima.

Así, pues, se trata de hallar los valores de a, b y c para los que la función $f(a,b,c) = \sum_{i=1}^{n}(ax_i + by_i + c - z_i)^2$ alcance el valor más pequeño posible, luego hace falta que se cumplan las siguientes condiciones:

$$\frac{\partial f}{\partial a} = 0, \frac{\partial f}{\partial b} = 0, \frac{\partial f}{\partial c} = 0 \tag{3.8}$$

$$\frac{\partial f}{\partial a} = \sum_{i=1}^{n} 2(ax_i + by_i + c - z_i)x_i = 0 \tag{3.9}$$

$$\frac{\partial f}{\partial b} = \sum_{i=1}^{n} 2(ax_i + by_i + c - z_i)x_i = 0 \tag{3.10}$$

$$\frac{\partial f}{\partial c} = \sum_{i=1}^{n} 2(ax_i + by_i + c - z_i)x_i = 0 \tag{3.11}$$

Despejando de (3.9),(3.10) y (3.11):

$$\sum_{i=1}^{n} ax_i^{\,2} + \sum_{i=1}^{n} bx_i y_i + \sum_{i=1}^{n} cx_i = \sum_{i=1}^{n} x_i z_i \tag{3.12}$$

$$\sum_{i=1}^{n} ax_i y_i + \sum_{i=1}^{n} by_i^{\,2} + \sum_{i=1}^{n} cy_i = \sum_{i=1}^{n} y_i z_i \tag{3.13}$$

$$\sum_{i=1}^{n} ax_i + \sum_{i=1}^{n} by_i + cn = \sum_{i=1}^{n} z_i \tag{3.14}$$

A partir de la solución de este sistema de ecuaciones se obtiene los valores de a, b y c.

El *kernel* lanzado comprueba cada uno de los píxeles considerados como vecinos iniciales, calcula su distancia euclídea y si es considerado como vecino real se añaden las coordenadas a los sumatorios de las ecuaciones anteriores para obtener el plano.

Cuando se comprueban todos los posibles vecinos, se resuelven las ecuaciones (3.12),(3.13) y (3.14) para obtener los valores de a, b y c.

$$c = \frac{\left[k_3 - \frac{\left(k_2 - \frac{k_1 \cdot k_6}{k_4}\right) \cdot \left(k_8 - \frac{k_6 \cdot k_7}{k_4}\right)}{k_5 - \frac{k_6^2}{k_4}} - \frac{k_1 \cdot k_7}{k_4}\right]}{n - \frac{k_1^2}{k_4} - \left[\frac{\left(k_2 - \frac{k_1 \cdot k_6}{k_4}\right)^2}{k_5 - \frac{k_6^2}{k_4}}\right]} \tag{3.15}$$

$$b = \frac{\left[\left(k_8 - \frac{k_6 \cdot k_7}{k_4}\right) - c \cdot \left(k_2 - \frac{k_1 \cdot k_6}{k_4}\right)\right]}{k_5 - \frac{k_6^2}{k_4}} \tag{3.16}$$

$$a = \frac{(k_{67} - k_6 \cdot b) - (k_1 \cdot c)}{k_4} \tag{3.17}$$

Donde

$$\begin{gathered} k_1 = \sum_{i=1}^{n} x_i \,; k_2 = \sum_{i=1}^{n} y_i \,; k_3 = \sum_{i=1}^{n} z_i \\ k_4 = \sum_{i=1}^{n} {x_i}^2 \,; k_5 = \sum_{i=1}^{n} {y_i}^2 \\ k_6 = \sum_{i=1}^{n} x_i y_i \,; k_7 = \sum_{i=1}^{n} x_i z_i \,;\; k_8 = \sum_{i=1}^{n} y_i z_i \end{gathered} \tag{3.18}$$

Todo este proceso se realiza en paralelo para cada uno de los píxeles de la imagen, logrando obtener todas las normales del plano en una sola ejecución del *kernel* correspondiente.

Finalmente la normal para cada píxel se define como:

$$\vec{n} = (a, b, -1) \tag{3.19}$$

Resumen de recursos

El *kernel* que obtiene la ecuación del plano emplea un número mayor de registros que el resto, siendo 30 los registros por hilo necesarios para la ejecución del mismo. Esto limita la ocupación por multiprocesador, ya que baja el número de *warps* activos por multiprocesador a 32 en vez de los 48 máximos que se consiguen para esta capacidad de cómputo.

Observando la Figura 3.6 se comprueba que se consigue el valor máximo de ocupación para varios valores de hilos por multiprocesador, eligiendo 1024 en este caso. En caso de haber elegido 512, la ocupación no cambiaría, pero se incrementaría el número de bloques, teniendo para el caso elegido 25 bloques en el *Grid*.

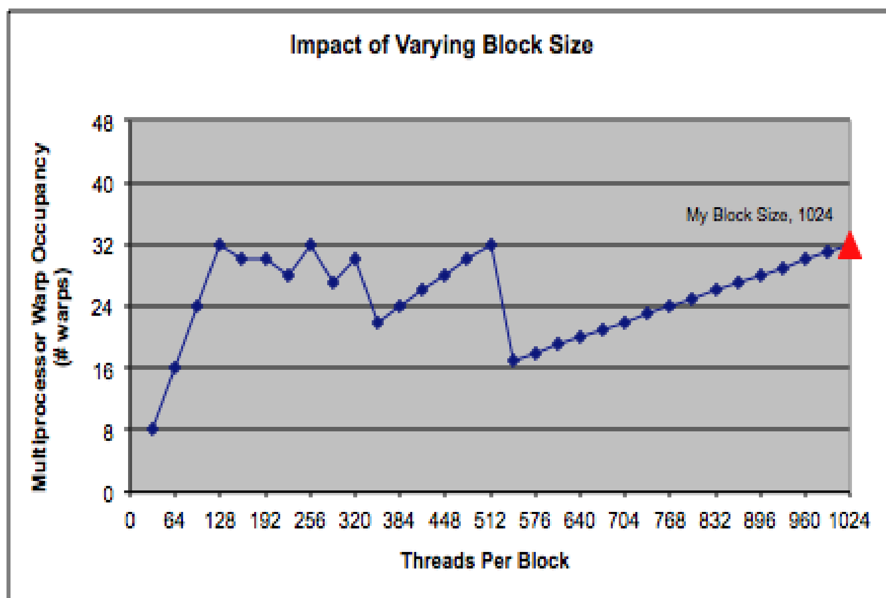

Figura 3.6: Impacto al variar el tamaño del bloque en *kernel2*

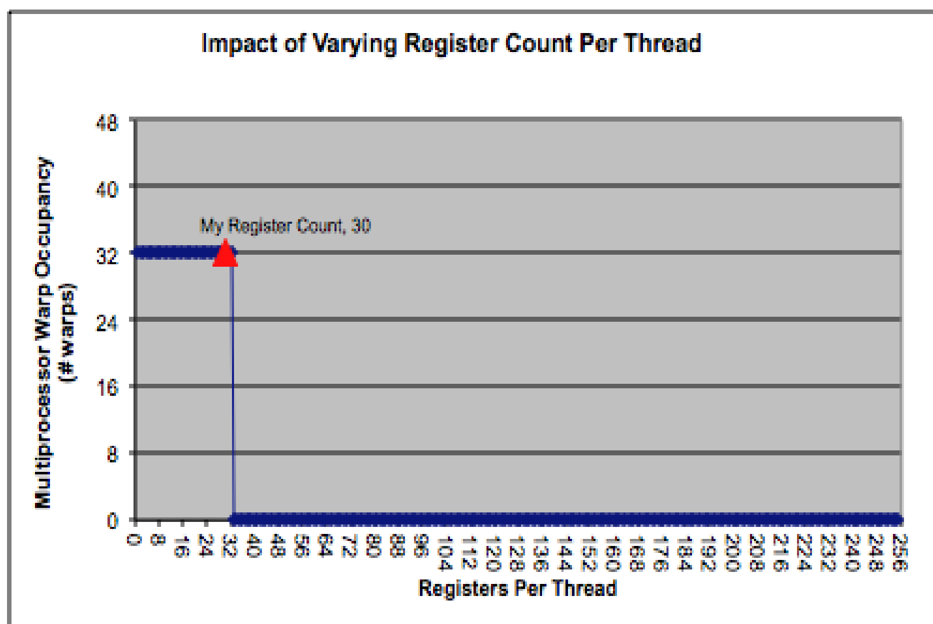

Figura 3.7: Impacto al variar el número de registro por hilo en *kernel2*

Sustituyendo en las ecuaciones descritas en 2.4.6

$$warps = \frac{\text{THREADS}}{threads_por_warp} = \frac{1024}{32} = 32 \tag{3.20}$$

$$limite_por_warps = \min\left(8, \frac{48}{32}\right) = \min(8{,}1.5) = 1.5 \tag{3.21}$$

$$reg_bloque = multiploSuperior(multiploSuperior(32{,}2) \cdot 30 \cdot 32, 512) = 30720 \tag{3.22}$$

$$limite_por_reg = multiploInferior\left(\frac{32768}{30720}\right) = 1 \tag{3.23}$$

$$bloques_activos_por_SM = \min(limite_por_warps, limite_por_reg) = \min(1.5, 1) = 1 \tag{3.24}$$

$$warps_{activos_{por_{SM}}} = 1 \cdot 32 = 32 \tag{3.25}$$

$$Occupancy = \frac{32}{48} = 0.67 \tag{3.26}$$

La Figura 3.8 muestra los resultados finales obtenidos para esta configuración. Como se comentaba antes no se consigue el 100% en la ocupación, siendo el valor máximo conseguido 67%.

2.) Enter your resource usage:	
Threads Per Block	1024
Registers Per Thread	30
Shared Memory Per Block (bytes)	0

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	1024
Active Warps per Multiprocessor	32
Active Thread Blocks per Multiprocessor	1
Occupancy of each Multiprocessor	67%

Figura 3.8: Resumen de recursos *kernel2*

3.4.3. Resumen de tiempos de cómputo de la función

En la Tabla 3.1 se muestran los resultados de tiempos obtenidos tras ejecutar la función con una imagen completa, obteniendo todas las normales para cada uno de los píxeles.

Kernel	**Tiempo (ms)**
Kernel1*(obtener núcleo de vecindad)*	0,078
Kernel2*(obtener plano)*	37,545
TOTAL	37,623

Tabla 3.1: Resumen de tiempos para el cálculo de la normal de todos los píxeles de una imagen

3.5. Cálculo de los diferenciales de área

La función *Calc_area* invoca al *kernel* que calcula los diferenciales de área. El diferencial de área se calcula como la suma de la cuarta parte del área de cada uno de los ocho triángulos que forma el píxel con sus vecinos más cercanos.

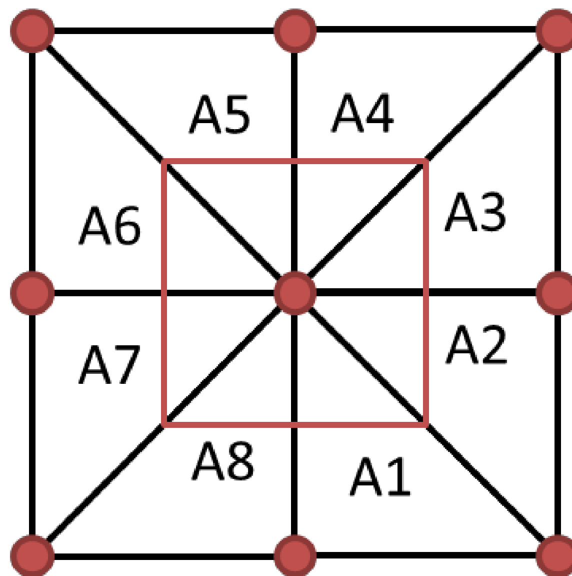

Figura 3.9: Diferencial de área

La ecuación del diferencial de área queda:

$$dA = \frac{1}{4}\sum_{n=1}^{8} A_n \tag{3.27}$$

Existen diferentes píxeles que por su localización en la imagen no contemplan todas las áreas de los triángulos, a continuación se muestra en la Tabla 3.2 con las singularidades para estos píxeles.

PÍXEL	**dA**
Esquina superior izquierda	$\frac{A1 + A2}{4}$
Esquina superior derecha	$\frac{A7 + A8}{4}$
Borde superior	$\frac{A1 + A2 + A7 + A8}{4}$
Borde lateral derecho	$\frac{A1 + A2 + A3 + A4}{4}$
Borde lateral izquierdo	$\frac{A5 + A6 + A7 + A7}{4}$
Esquina inferior derecha	$\frac{A6 + A5}{4}$
Esquina inferior izquierda	$\frac{A1 + A2}{4}$
Borde inferior	$\frac{A3 + A4 * A5 + A6}{4}$

Tabla 3.2: Diferencial de área para cada píxel

La función sólo cuenta como parámetro de entrada las coordenadas 3D de cada píxel, y su salida es el diferencial de área obtenido para cada uno de ellos. Al ejecutar el *kernel*, se obtienen de forma paralela todos los diferenciales de área de la imagen. Dicha llamada al *kernel* se realiza pasando como parámetro los mismos que la función *Calc_area* y definiendo el número de *Grid* y *Block* para dicho *kernel*.

Resumen de recursos

Como se indicó anteriormente la tarjeta trabaja con una capacidad de cómputo de 2.1, para la función programada se obtienen 15 registros por hilo. Conociendo este dato es posible establecer un número de bloques fijo con el que se consiga el mayor rendimiento posible en la tarjeta.

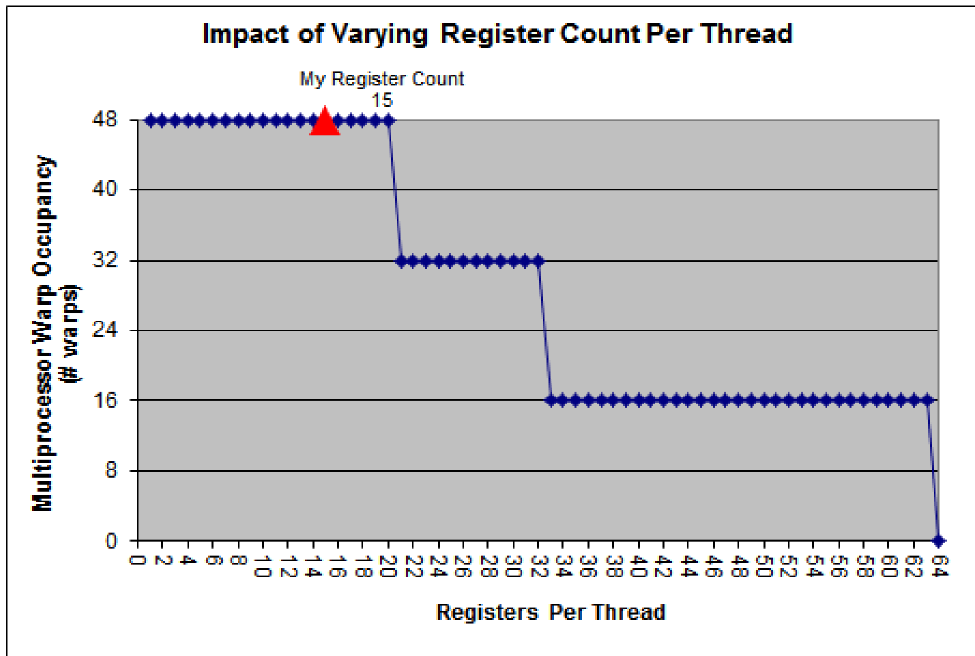

Figura 3.10: Impacto al variar el número de registro por hilo en área

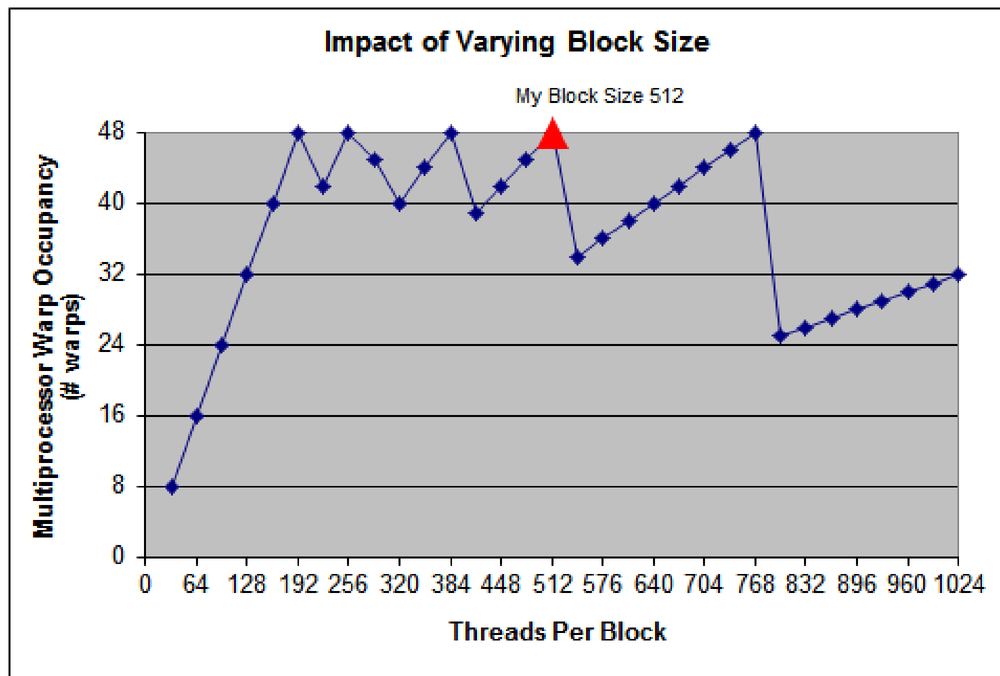

Figura 3.11: Impacto al variar el tamaño del bloque en área

Observando las gráficas anteriores se concluye que para un número de registros inferior a 20, se consigue máximo rendimiento para distintos tamaños de bloque, utilizando para este caso 512, por lo que los *Grid* necesarios ascienden a 149. Para este *kernel* no se utiliza memoria compartida, no influyendo en el cálculo de la ocupación.

Sustituyendo en las ecuaciones descritas en 2.4.6

$$warps = \frac{\text{THREADS}}{threads_por_warp} = \frac{512}{32} = 16 \tag{3.28}$$

$$limite_por_warps = \min\left(8, \frac{48}{16}\right) = 3 \tag{3.29}$$

$$reg_bloque = multiploSuperior(multiploSuperior(16,2) \cdot 15 \cdot 32, 512) = 7680 \tag{3.30}$$

$$limite_por_reg = multiploInferior\left(\frac{32768}{7680}\right) = 4 \tag{3.31}$$

$$bloques_activos_por_SM = \min(limite_por_warps, limite_por_reg) = \min(3,4) = 3 \tag{3.32}$$

$$warps_{activos_{por_{SM}}} = 3 \cdot 16 = 48 \tag{3.33}$$

$$Occupancy = \frac{48}{48} = 1 \tag{3.34}$$

La Figura 3.12 certifica que se obtiene una ocupación del 100%, consiguiendo así que la ejecución paralela sea máxima.

2.) Enter your resource usage:	
Threads Per Block	512
Registers Per Thread	15
Shared Memory Per Block (bytes)	0

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	1536
Active Warps per Multiprocessor	48
Active Thread Blocks per Multiprocessor	3
Occupancy of each Multiprocessor	100%

Figura 3.12: Resumen de recursos área

3.5.1. Resumen de tiempos de cómputo de la función

El resumen de tiempos final para la obtención de los diferenciales de área para cada píxel de la imagen se muestra en la Tabla 3.3

Kernel	**Tiempo (ms)**
Área	1,84
TOTAL	**1,84**

Tabla 3.3: Resumen de tiempos cálculo de área

3.6. Cálculo de *rho* relativo

Este conjunto de funciones se encargan de estimar el coeficiente relativo de absorción (*rho*) de cada punto de la escena bajo estudio.

3.6.1. Obtener *rho*

Este *kernel* calcula los *rho* de cada píxel de la imagen debidos a su normal y su diferencial de área mediante la expresión (3.35).

$$\rho_i = a_i^{origen} \frac{d_i^4}{dA_i \cos\theta} \tag{3.35}$$

Se redireccionan los puntos en la dirección del rayo óptico y se desplazan al origen, obteniendo el ángulo que forma la normal del píxel con el rayo incidente y la distancia entre cámara y píxel como se muestra en la Figura 3.13,

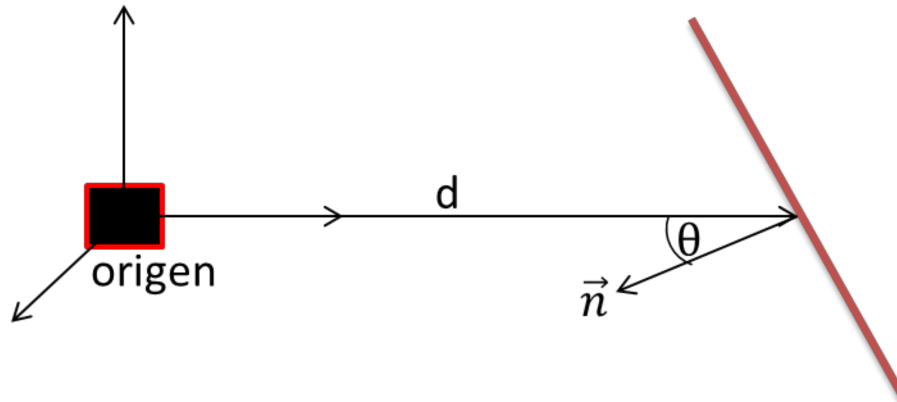

Figura 3.13: Trazado de rayos para obtener *rho*

donde *d* es la distancia del origen al píxel bajo estudio, θ es el ángulo formado entre el rayo incidente de la cámara y la normal al plano del píxel. *dA* es el diferencial de área y *a* la amplitud de dicho píxel.

Los parámetros de entrada a esta función son las coordenadas 3D y la amplitud de cada píxel de la imagen, así como la normal obtenida para cada uno de ellos. El dato de salida es la *rho* estimada.

Sabiendo que en todo momento el origen está establecido por la cámara y que este es (0,0,0), se obvia en la programación de esta función consiguiendo así una mejora en el tiempo de cómputo bastante apreciable, dicha mejora se muestra en la Tabla 3.4.

Rho3 considerando origen como (0,0,0)	191,296 *us*
Rho3 obviando origen	141,568 *us*

Tabla 3.4: Comparativa de tiempos de *rho*

Puede verse que se consigue reducir el tiempo de cómputo en 50 *us*, lo que supone una reducción superior al 25%. Estos tiempos son relativos a la parte del cálculo de *rho*, todavía sin normalizar.

Resumen de recursos

Esta parte utiliza 15 registros por hilo y 0 bytes en memoria compartida, obteniendo las siguientes gráficas de rendimiento.

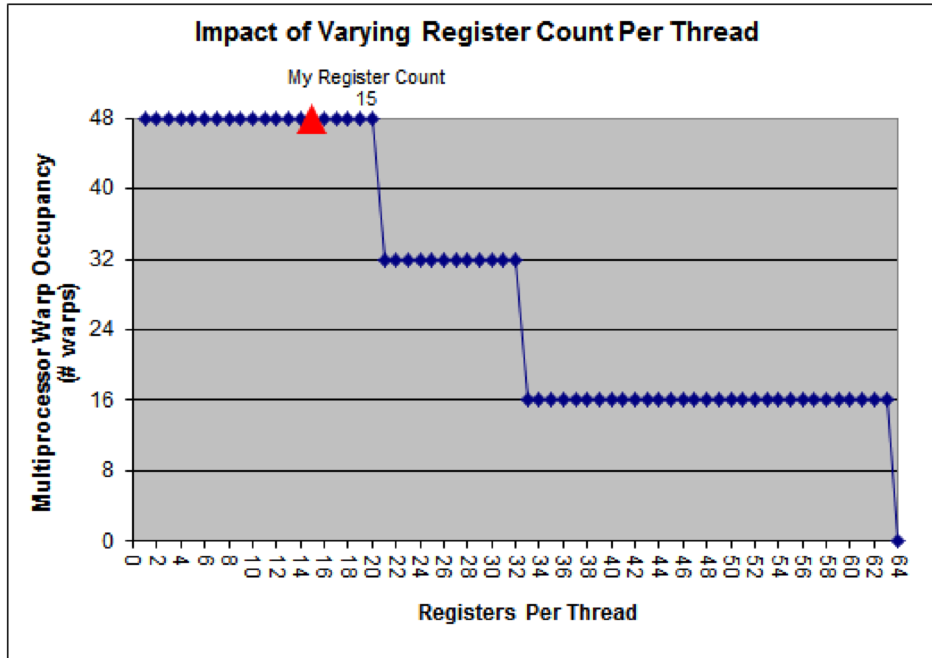

Figura 3.14: Impacto al variar el número de registro por hilo en *rho3*

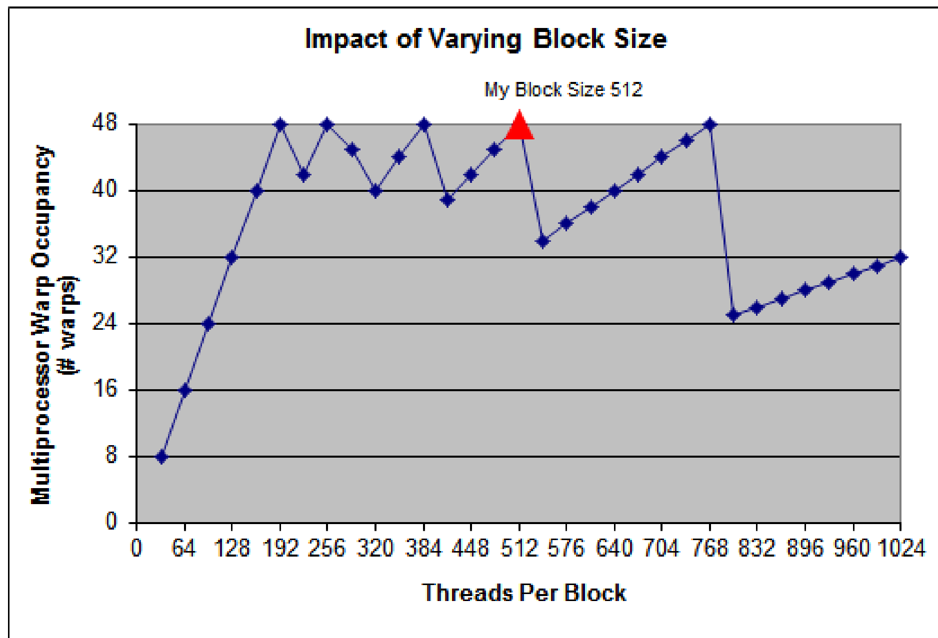

Figura 3.15: Impacto al variar el tamaño del bloque en *rho3*

Sustituyendo en las ecuaciones descritas en 2.4.6

$$warps = \frac{\text{THREADS}}{threads_por_warp} = \frac{512}{32} = 16 \tag{3.36}$$

$$limite_por_warps = \min\left(8, \frac{48}{16}\right) = 3 \tag{3.37}$$

$$\begin{aligned} reg_bloque &= multiploSuperior(multiploSuperior(16,2) \cdot 15 \cdot 32, 512) \\ &= 7680 \end{aligned} \tag{3.38}$$

$$limite_por_reg = multiploInferior\left(\frac{32768}{7680}\right) = 4 \tag{3.39}$$

$$\begin{aligned} &bloques_activos_por_SM = \\ &\min(limite_por_warps, limite_por_reg) = \min(3, 4) = 3 \end{aligned} \tag{3.40}$$

$$warps_{activos_{por_{SM}}} = 3 \cdot 16 = 48 \tag{3.41}$$

$$Occupancy = \frac{48}{48} = 1 \tag{3.42}$$

De igual forma que para las funciones anteriores, se consigue una ocupación máxima por debajo de 20 registro por hilo, eligiendo para este caso 512 bloques por *Grid*. Los resultados obtenidos se muestran en la Figura 3.16.

2.) Enter your resource usage:	
Threads Per Block	512
Registers Per Thread	15
Shared Memory Per Block (bytes)	0

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	1536
Active Warps per Multiprocessor	48
Active Thread Blocks per Multiprocessor	3
Occupancy of each Multiprocessor	100%

Figura 3.16: Resumen de recursos *rho3*

3.6.2. Cálculo del máximo de un conjunto de valores en paralelo

Teniendo calculados los *rho*, es necesario buscar el valor máximo entre todos ellos para normalizar y obtener el *rho* relativo. Para ello se utiliza un algoritmo de reducción en paralelo adaptado para obtener el máximo de cada bloque.

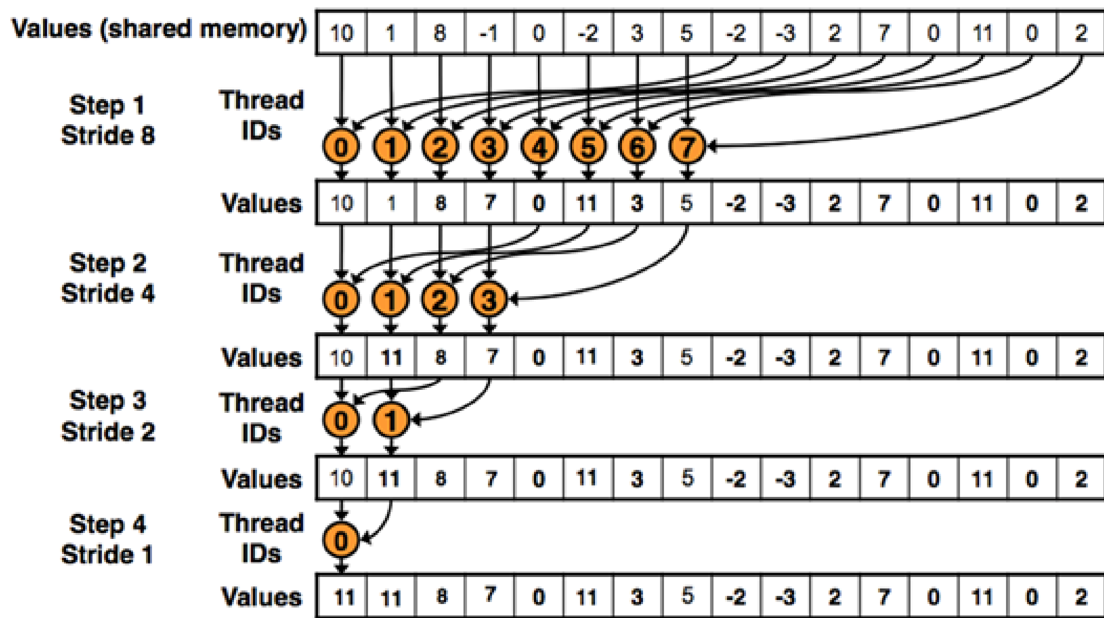

Figura 3.17: Reducción paralela con direccionamiento secuencial

Como se muestra en la Figura 3.17 se empleará un direccionamiento secuencial, donde se obtendrá el máximo entre dos valores concretos del *array*. Para mejorar el rendimiento y los tiempos de cómputo, conociendo el número de iteraciones en un tiempo de compilación, es posible realizar un desenrollado completo del bucle, mejorando así la ejecución paralela de los elementos. Esto también se facilita si el tamaño del bloque es fijo.

Combinando reducciones paralelas y secuenciales se consigue aún mejor rendimiento, cada hilo carga y obtiene el valor máximo de múltiples elementos en memoria compartida, utilizando reducción basada en árbol.

Es posible mejorar aún más el rendimiento:

- Mejorar la latencia con más trabajos por hilo
- Mas hilos por bloque reduce el nivel en el árbol de invocaciones de *kernel* recursivos
- Sobrecarga del lanzamiento de un *kernel* en los últimos niveles con pocos bloques

Utilizando este método se obtiene el máximo de los valores del bloque, por lo que se tendrá un máximo por bloque, haciendo necesario volver a aplicar otro algoritmo similar para obtener un único valor máximo. Poniendo como ejemplo el caso que nos ocupa, para bloques de 512 hilos, con una dimensión total de ancho*alto de la imagen (176*144), se obtiene un número de bloques igual a 50, esto quiere decir que cada bloque obtendrá el máximo de los 512 elementos que en él se almacenan, por lo que se tendrán 50 máximos en la primera fase.

Como se mencionaba anteriormente, es necesario volver a aplicar otro algoritmo de obtención del máximo que realice la misma operación y conseguir un único máximo. Este segundo *kernel* sólo necesita de un bloque con un número de hilos mayor al número de máximos anterior, por mejora de rendimiento se eligen 256 hilos por bloque.

Resumen de recursos

A continuación se muestran las correspondientes figuras de la ocupación de cada *kernel*, así como el número de registros necesarios y el tamaño de memoria compartida.

El *kernel reduce6* utiliza 18 registros por hilo, dentro del límite de los 20 que permiten máxima eficiencia y un tamaño de memoria compartida de 2048 bytes. Con una configuración de 512 hilos por bloque se consigue máxima ocupación por multiprocesador.

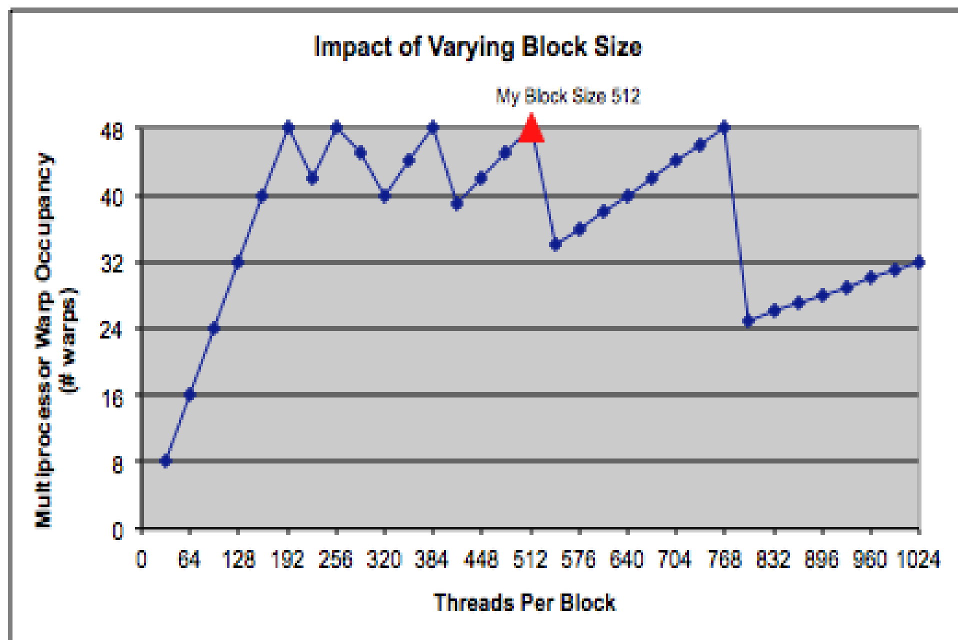

Figura 3.18: Impacto al variar el tamaño del bloque en *reduce6*

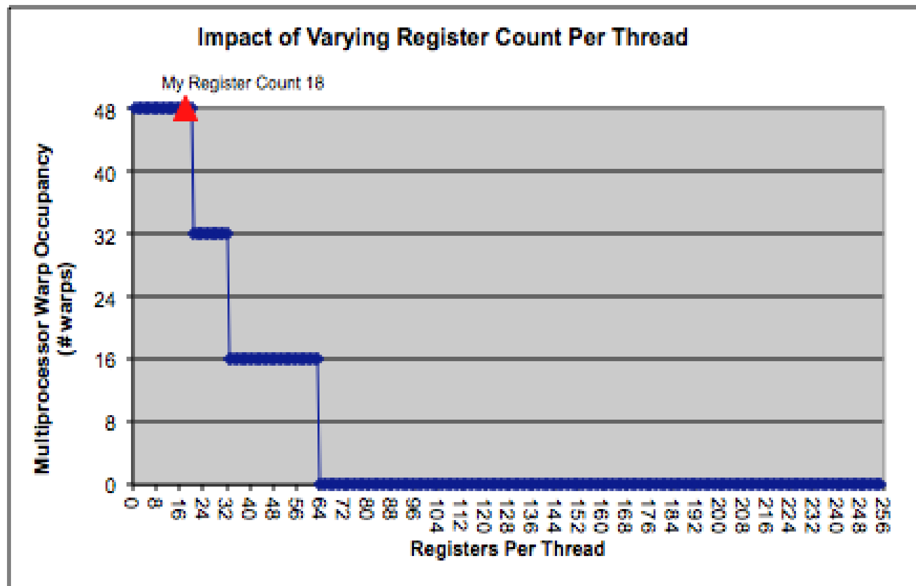

Figura 3.19: Impacto al variar el número de registro por hilo en *reduce6*

Para la ejecución de este *kernel* se necesita utilizar la memoria compartida por todos los hilos. La Figura 3.20 muestra los *warps* que serán empleados dependiendo del número de bytes que sean utilizados. En este caso es necesaria poca memoria compartida, por lo que no afecta a la hora de obtener la ocupación máxima posible.

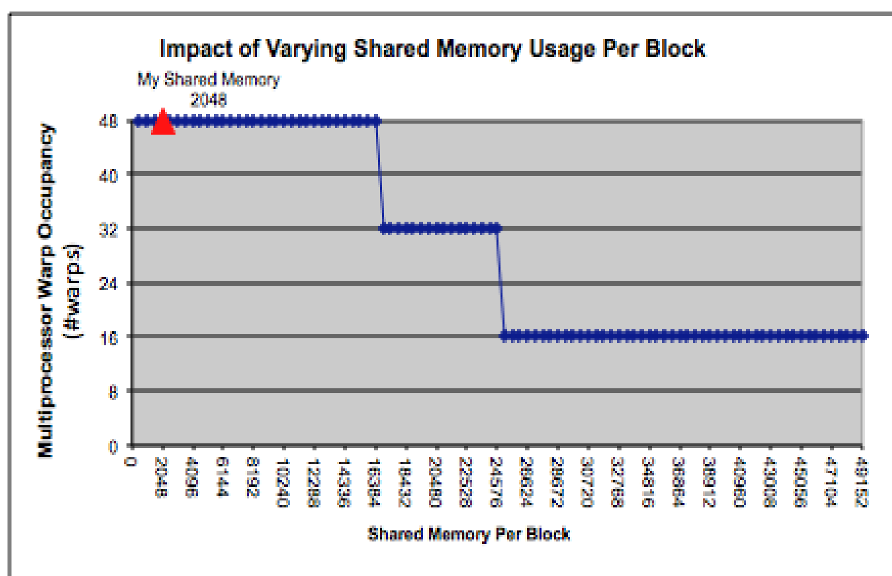

Figura 3.20: Impacto al variar la memoria compartida usado por bloque en *reduce6*

Sustituyendo en las ecuaciones descritas en 2.4.6

$$warps = \frac{\text{THREADS}}{threads_por_warp} = \frac{512}{32} = 16 \qquad (3.43)$$

$$limite_por_warps = \min\left(8, \frac{48}{16}\right) = 3 \quad (3.44)$$

$$limite_{por_{shared}} = multiploInferior\left(\frac{49152}{2048}\right) = 24 \quad (3.45)$$

$$reg_bloque = multiploSuperior(multiploSuperior(16,2) \cdot 18 \cdot 32, 512) = 9216 \quad (3.46)$$

$$limite_por_reg = multiploInferior\left(\frac{32768}{9216}\right) = 3 \quad (3.47)$$

$$bloques_activos_por_SM = \min(limite_por_warps, limite_por_reg, limite_por_shared) = \min(3, 3,24) = 3 \quad (3.48)$$

$$warps_{activos_{por_{SM}}} = 3 \cdot 16 = 48 \quad (3.49)$$

$$Occupancy = \frac{48}{48} = 1 \quad (3.50)$$

Finalmente se obtiene el valor de ocupación calculado anteriormente incluyendo el número de bytes de la memoria compartida que se utilizan.

2.) Enter your resource usage:	
Threads Per Block	512
Registers Per Thread	18
Shared Memory Per Block (bytes)	2048

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	1536
Active Warps per Multiprocessor	48
Active Thread Blocks per Multiprocessor	3
Occupancy of each Multiprocessor	100%

Figura 3.21: Resumen de recursos *reduce6*

El *kernel reduce3* trabaja con un número menor de datos, por lo que sus dimensiones y su número de registros se reducen, siendo necesarios solamente 9 registros por hilo y un único bloque con 256 para obtener solución válida. La Figura 3.22, Figura 3.23 y la Figura 3.24 muestran el impacto que tiene en el número de *warps* variar los hilos por bloque con los registro mencionados anteriormente y con una memoria compartida de 2048 bytes.

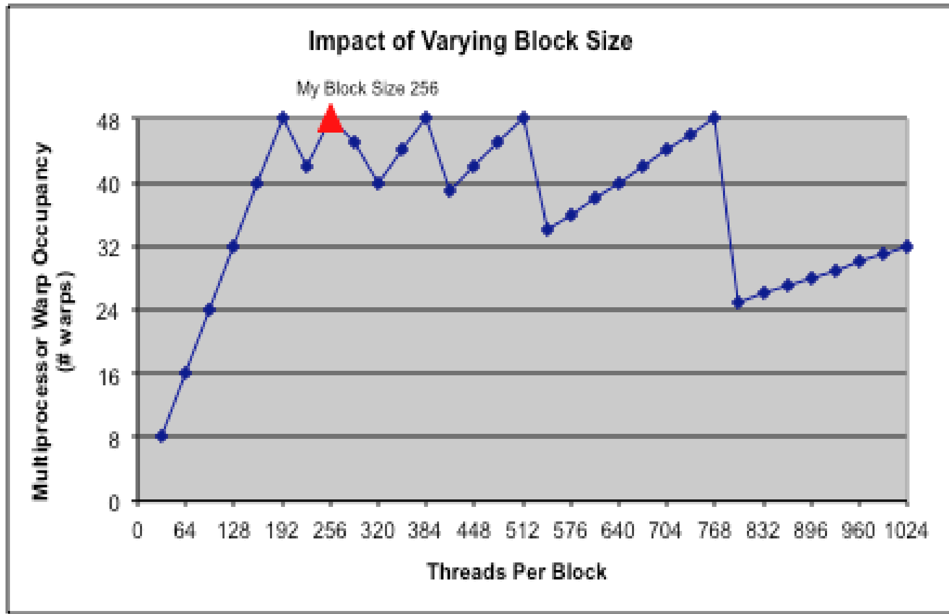

Figura 3.22: Impacto al variar el tamaño del bloque en *reduce3*

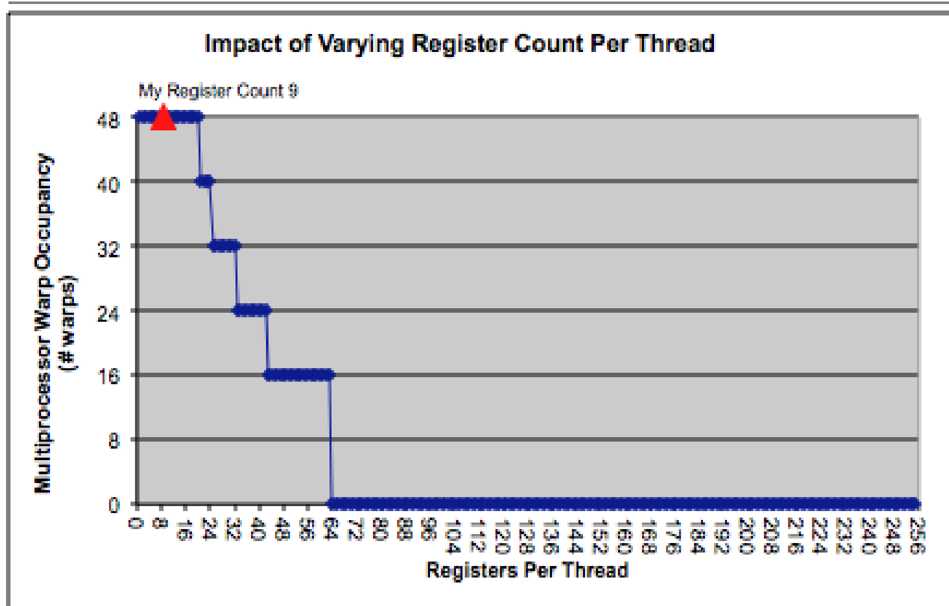

Figura 3.23: Impacto al variar el número de registro por hilo en *reduce3*

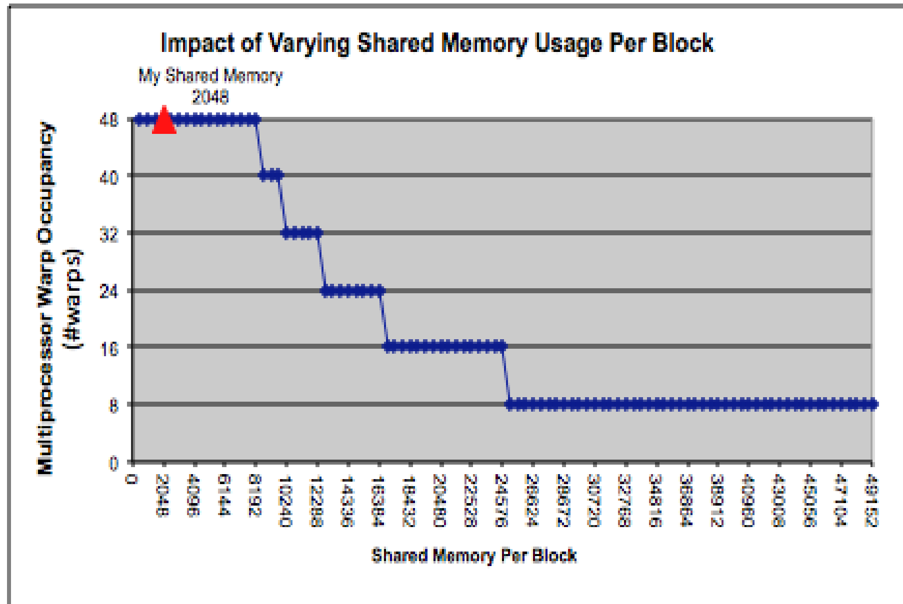

Figura 3.24: Impacto al variar la memoria compartida usado por bloque en *reduce3*

Sustituyendo en las ecuaciones descritas en 2.4.6

$$warps = \frac{\text{THREADS}}{threads_por_warp} = \frac{256}{32} = 8 \tag{3.51}$$

$$limite_por_warps = \min\left(8, \frac{48}{8}\right) = 6 \tag{3.52}$$

$$limite_{por_{shared}} = multiploInferior\left(\frac{49152}{2048}\right) = 24 \tag{3.53}$$

$$reg_bloque = multiploSuperior(multiploSuperior(8{,}2) \cdot 9 \cdot 32, 512) = \\ = 2560 \tag{3.54}$$

$$limite_por_reg = multiploInferior\left(\frac{32768}{2560}\right) = 12 \tag{3.55}$$

$$bloques_activos_por_SM = \\ \min\left(limite_{por_{warps}}, limite_{por_{reg}}, limite_{por_{shared}}\right) = \\ = \min(6{,}12\,{,}24) = 6 \tag{3.56}$$

$$warps_{activos_{por_{SM}}} = 6 \cdot 8 = 48 \tag{3.57}$$

$$Occupancy = \frac{48}{48} = 1 \tag{3.58}$$

La ocupación vuelve a ser máxima para este *kernel* con la configuración elegida como se observa en la Figura 3.25.

2.) Enter your resource usage:	
Threads Per Block	256
Registers Per Thread	9
Shared Memory Per Block (bytes)	2048

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	1536
Active Warps per Multiprocessor	48
Active Thread Blocks per Multiprocessor	6
Occupancy of each Multiprocessor	100%

Figura 3.25: Resumen de recursos *reduce3*

3.6.3. Cálculo de *rho* relativo

Teniendo los valores de *rho* y el máximo valor de ésta, sólo queda normalizar. Para ello se utiliza un *kernel* que únicamente divida cada valor de *rho* calculado inicialmente, entre el valor máximo obtenido como se muestra en (3.59).

$$\rho_i^{rel} = \frac{1}{\max\{\rho_i\}} \left(a_i^{origen} \frac{d_i^4}{dA_i \cos\theta} \right) \tag{3.59}$$

Resumen de recursos

Observando la Figura 3.26y la Figura 3.27, se demuestra que se consigue el mayor número de *warps* teniendo 12 registros por hilo, para una configuración de 512 hilos por bloque. En este *kernel* no es necesario utilizar memoria compartida, por lo que no afecta al cálculo de la ocupación.

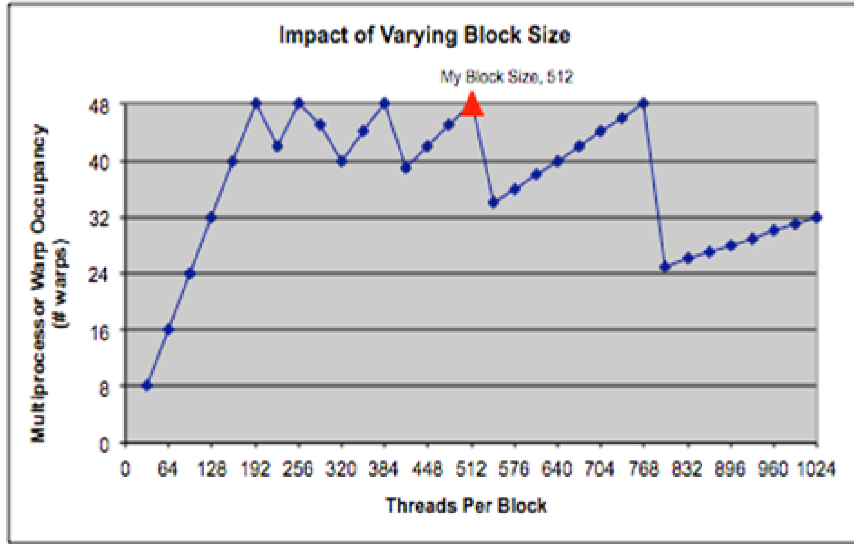

Figura 3.26: Impacto al variar el tamaño del bloque en *rho4*

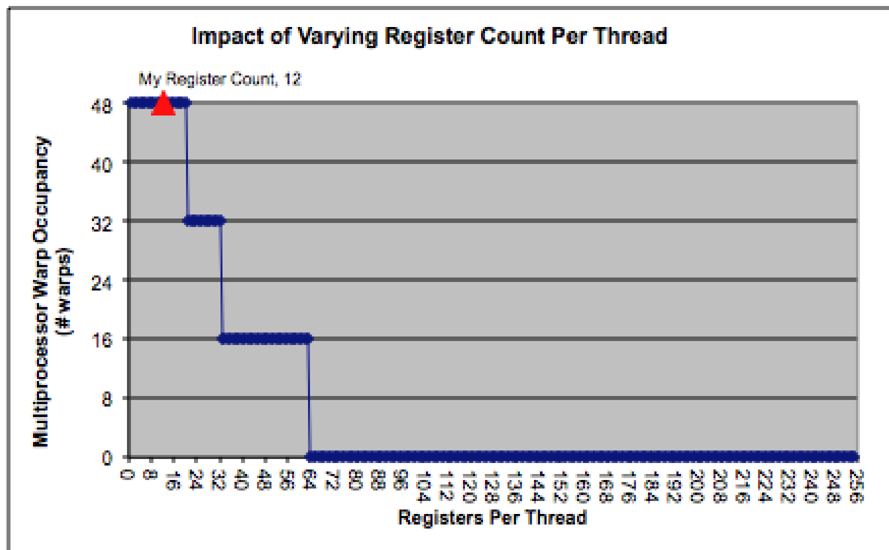

Figura 3.27: Impacto al variar el número de registro por hilo en *rho4*

Sustituyendo en las ecuaciones descritas en 2.4.6

$$warps = \frac{\text{THREADS}}{threads_por_warp} = \frac{512}{32} = 16 \quad (3.60)$$

$$limite_por_warps = \min\left(8, \frac{48}{16}\right) = 3 \quad (3.61)$$

$$reg_bloque = multiploSuperior(multiploSuperior(16{,}2) \cdot 12 \cdot 32, 512) = \\ = 6144 \quad (3.62)$$

$$limite_por_reg = multiploInferior\left(\frac{32768}{6144}\right) = 5 \quad (3.63)$$

$$bloques_activos_por_SM = \min(3,5) = 3 \quad (3.64)$$

$$warps_activos_por_SM = 3 \cdot 16 = 48 \quad (3.65)$$

$$Occupancy = \frac{48}{48} = 1 \quad (3.66)$$

La ocupación obtenida por CUDA occupancy calculator es igual a la obtenida teóricamente como se muestra en la Figura 3.28.

2.) Enter your resource usage:	
Threads Per Block	512
Registers Per Thread	12
Shared Memory Per Block (bytes)	0

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	1536
Active Warps per Multiprocessor	48
Active Thread Blocks per Multiprocessor	3
Occupancy of each Multiprocessor	100%

Figura 3.28: Resumen de recursos *rho4*

3.6.4. Resumen de tiempos de cómputo de la función

El resumen de tiempos final para la obtención de *rho* se muestra en la Tabla 3.5.

Kernel	**Tiempo (us)**
Rho3(obtener rho)	141,568
Reduce6(cálculo del máximo)	100,83
Reduce3(cálculo del máximo)	8,64
Rho4(obtener rho relativo)	32,28
TOTAL	**283,318**

Tabla 3.5: Resumen de tiempos del cálculo de *rho*

Capítulo 4.Resultados

4.1. Resultados experimentales

En este capítulo se muestran los resultados experimentales obtenidos. Todas las pruebas experimentales han sido realizadas con imágenes reales adquiridas cono la cámaras *ToF* SR4000 de Mesa-Imaging, en un ordenador cuyas características hardware y software se exponen en el Anexo 1.

4.1.1. Efecto del número de vecinos en el cálculo de normales

Para realizar el cálculo de las normales pueden elegirse los vecinos más cercanos con un radio determinado. La elección de dicho radio implica que se tenga un número menor o mayor de vecinos iniciales. En la práctica, se consideran radios de vecindad de uno o dos píxeles por motivos de eficiencia

En este apartado se comparan los resultados obtenidos con cada uno de estos radios, visualizando en ambos casos los vectores normales calculados.

La Figura 4.1 muestra la imagen utilizada para el cálculo de normales. A continuación, la Figura 4.2 y la Figura 4.3 presentan las normales obtenidas para los distintos grados de vecindad.

Figura 4.1: Imagen para el estudio de las normales

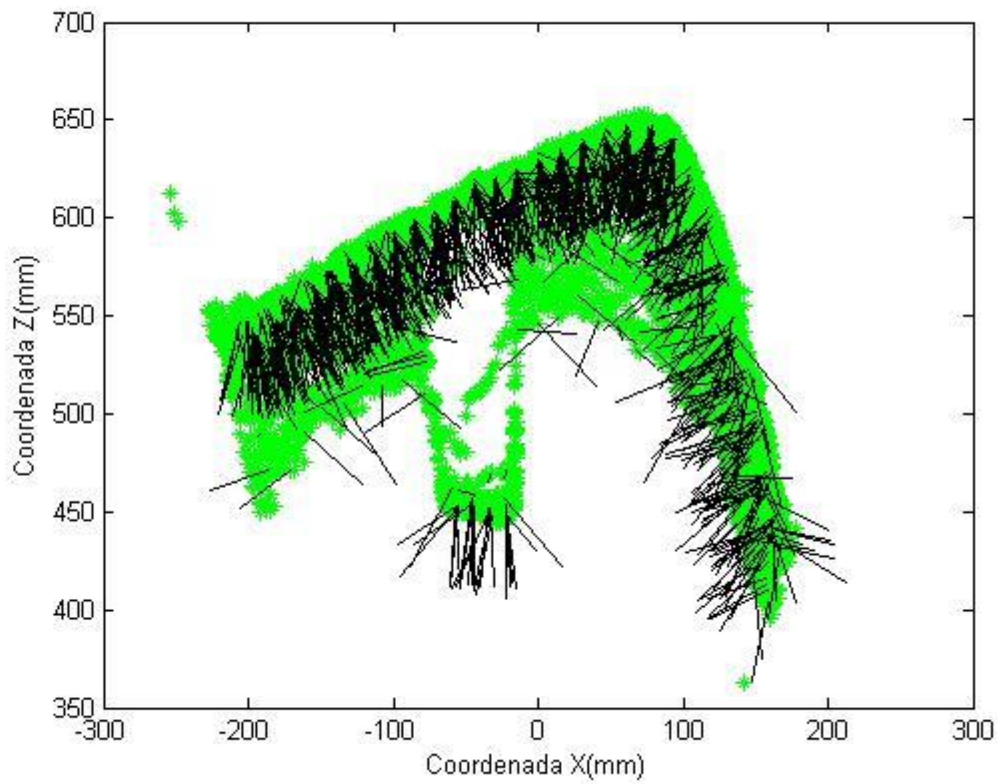

Figura 4.2: Normales obtenidas con radio 1

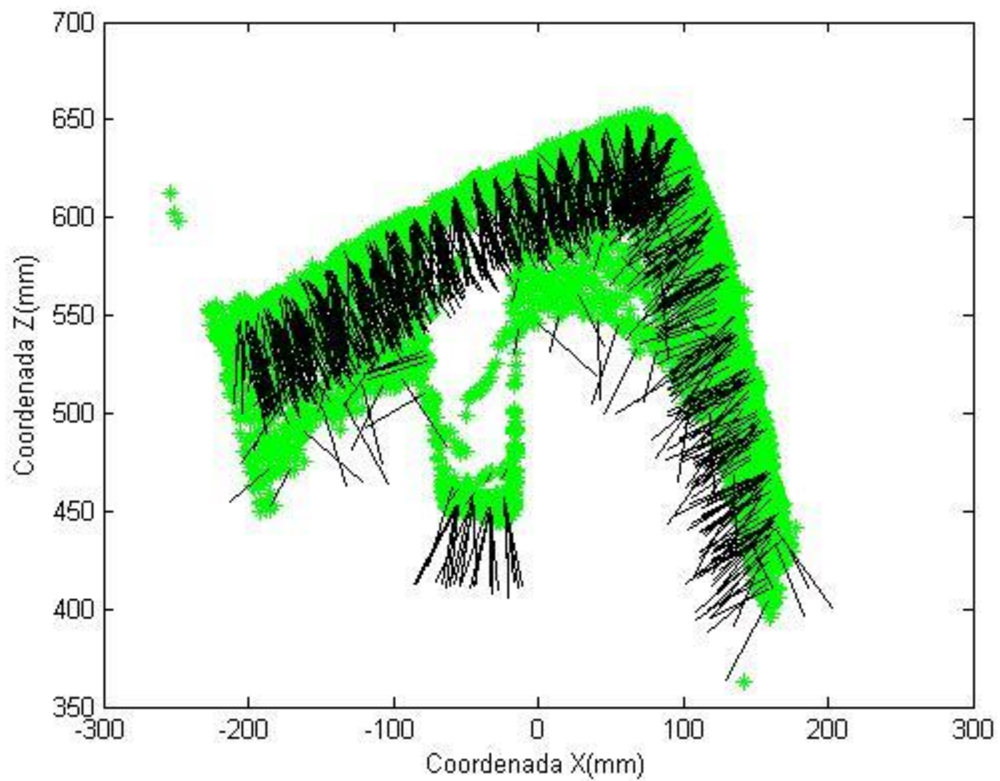

Figura 4.3: Normales obtenidas con radio 2

Observando la Figura 4.2 y la Figura 4.3 se concluye que las normales obtenidas con radio igual a dos tienen una mayor precisión que las calculadas con radio 1, ya que consideran un mayor número de píxeles vecinos para obtener la ecuación del plano, y el correspondiente vector normal.

Los tiempos de ambas funciones se muestran en el apartado 4.2, cabe mencionar aquí que para un radio menor los tiempos de ejecución son menores, de esta forma, será necesario alcanzar un compromiso entre precisión y tiempo de cómputo.

4.1.2. Resultado final: cálculo de normales, diferencial de área y coeficiente de absorción relativo (rho)

Con los datos obtenidos con el programa en CUDA se representan las figuras obtenidas de las normales, el diferencial de áreas y el coeficiente de absorción relativo para diferentes imágenes.

Imagen 1

La primera imagen utilizada contiene un patrón de cuadros con un cierto ángulo para poder apreciar mejor el efecto de las normales.

Figura 4.4: Amplitud de la imagen 1

Las normales obtenidas para la imagen se muestran en la figura. Es difícil apreciar la orientación de cada una de ellas, pero puede observarse que la gran mayoría apunta hacia la cámara. Las normales obtenidas en los bordes de la imagen no pueden considerarse del todo correctas, ya que la orientación puede diferir bastante de la real, al tener un número inferior de vecinos que los píxeles interiores.

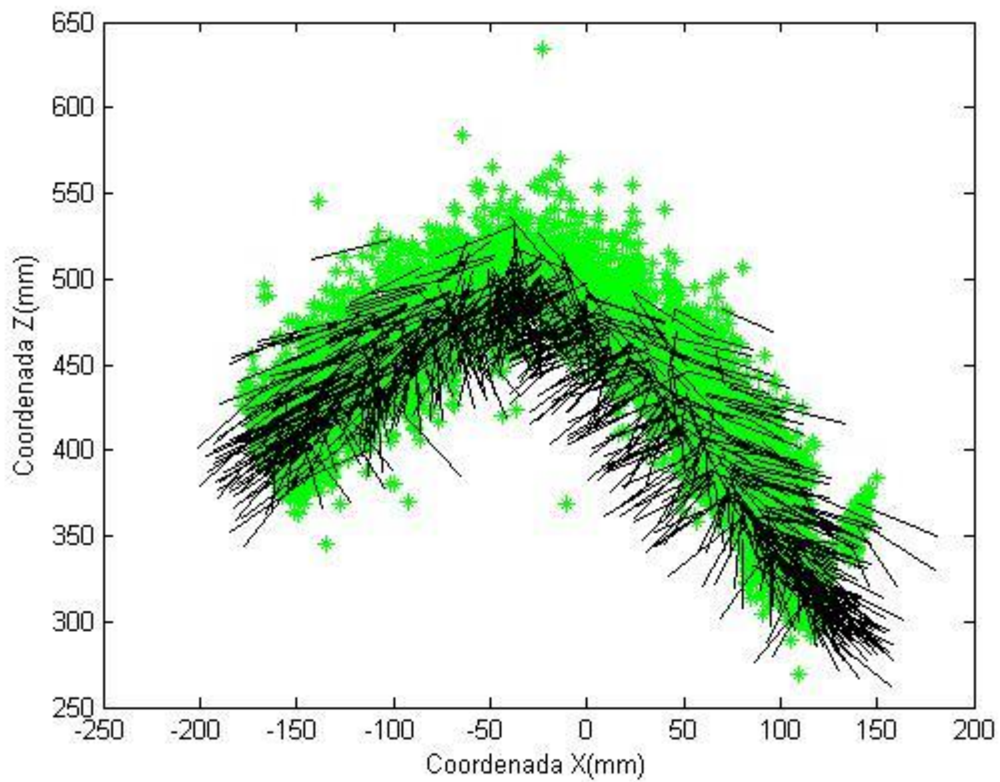

Figura 4.5: Normales de la imagen 1

Representando los datos obtenidos para el diferencial de área se observa que para esta imagen se distingue claramente el patrón de cuadros utilizado.

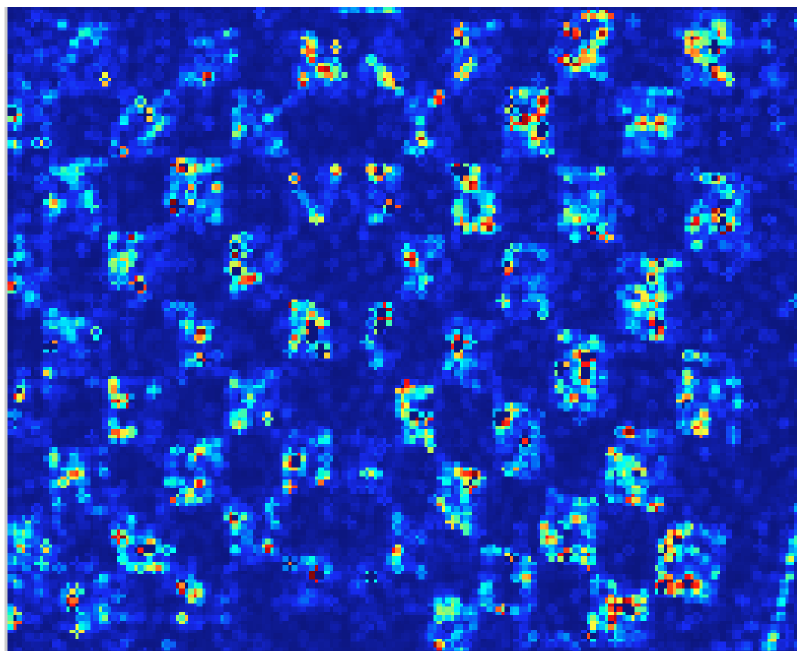

Figura 4.6: Diferencial de área de la imagen 1

De la misma forma que para el diferencial de área, el resultado de *rho* se muestra en la figura, donde también es posible distinguir el patrón de la imagen.

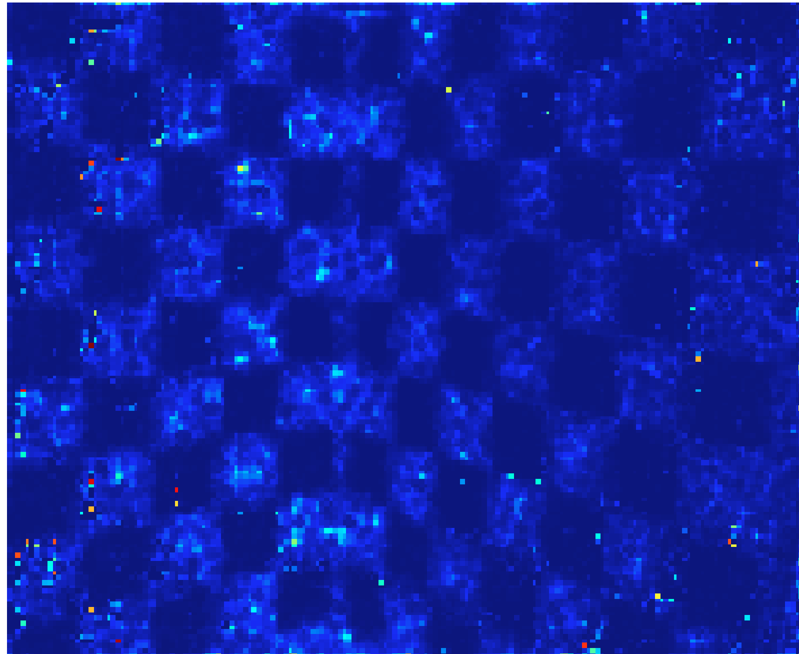

Figura 4.7: *Rho* de la imagen 1

Imagen 2

La segunda imagen de prueba contiene una caja situada en la intersección de dos paredes.

Figura 4.8: Amplitud de la imagen 2

De igual forma que en la imagen anterior es difícil apreciar con claridad todas las normales conseguidas. Si es posible ver que las normales referidas a la pared de fondo son puntos, por lo que la normal aquí es correcta. De forma semejante, las normales de la pared lateral tienen su vector perpendicular a ella. En los laterales de la imagen no es posible determinar de forma correcta las normales

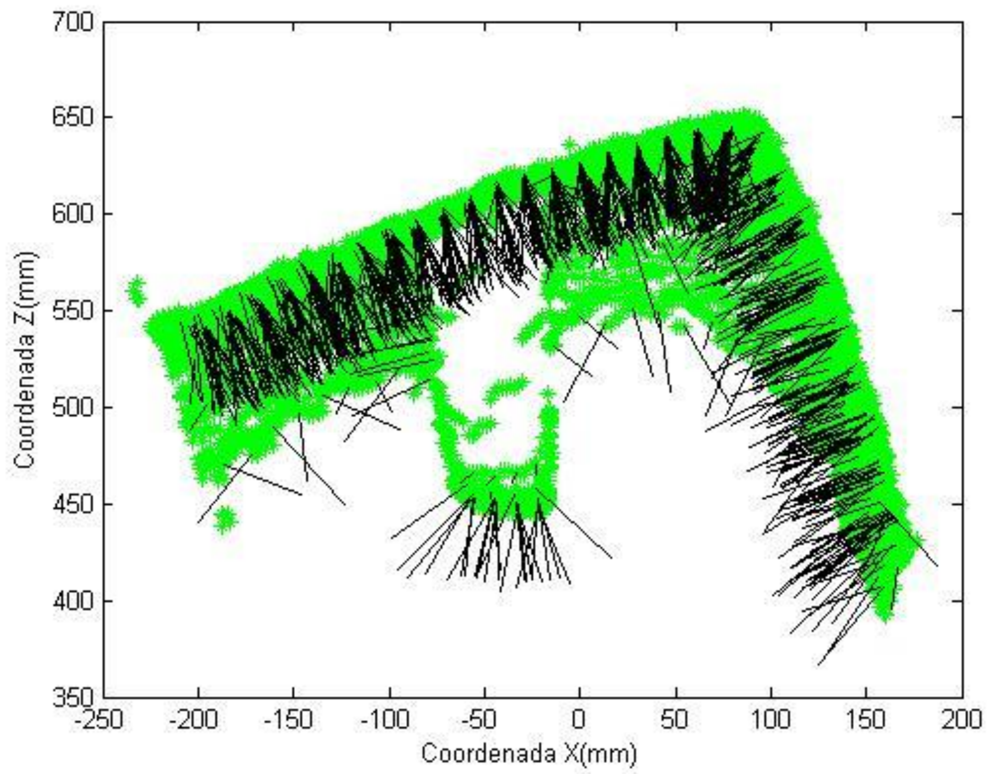

Figura 4.9: Normales de la imagen 2

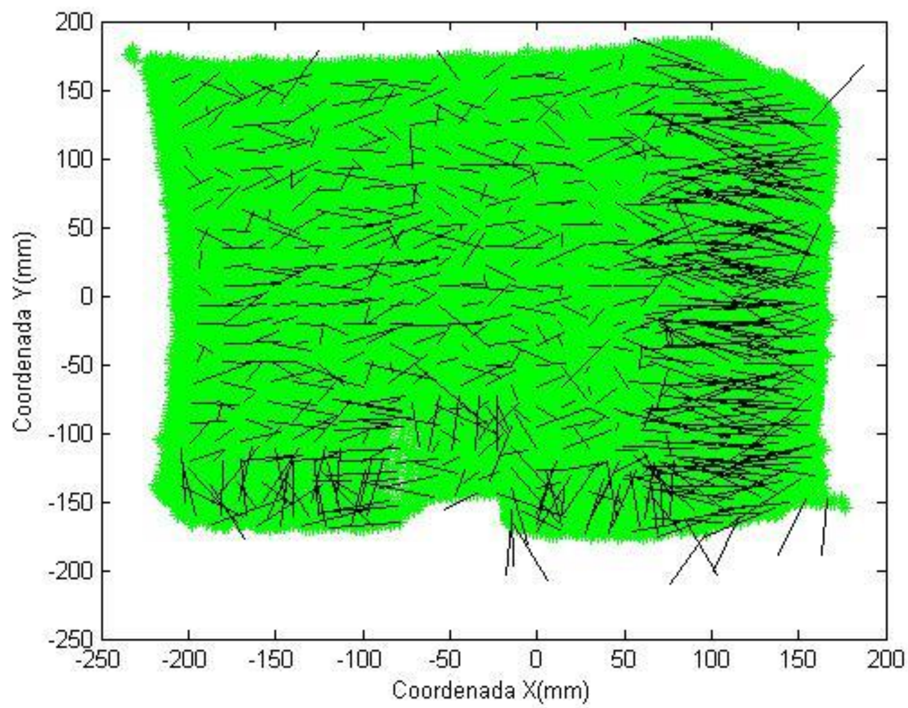

Figura 4.10: Normales de la imagen 2

En la figura del diferencial de áreas, se observa la forma de la caja situada entre ambas paredes, lugar donde el diferencial de áreas es mayor por la dispersión de puntos.

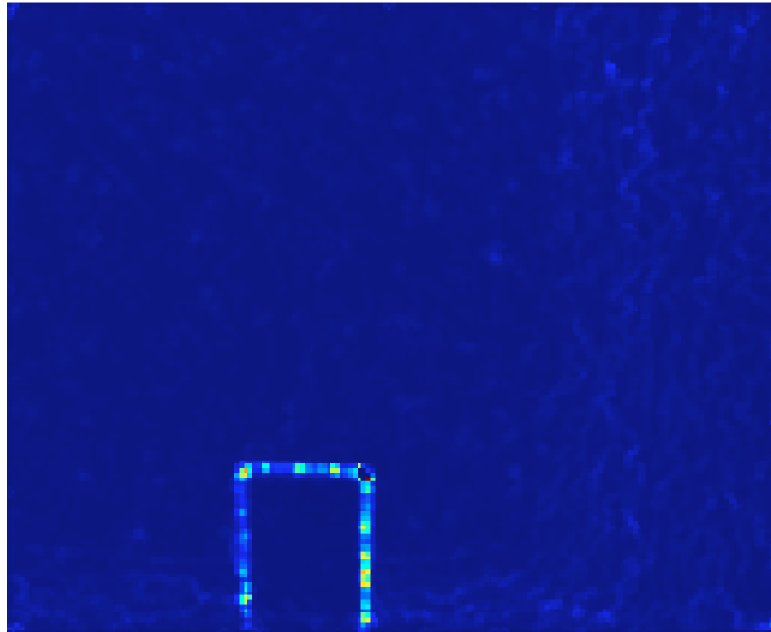

Figura 4.11: Diferencial de área de la imagen 2

La imagen de *rho* muestra en tonos más claros aquellos píxeles cuya normal apunta directamente a la cámara, siendo así la pared de fondo y el frontal de la caja las zonas más claras de la imagen.

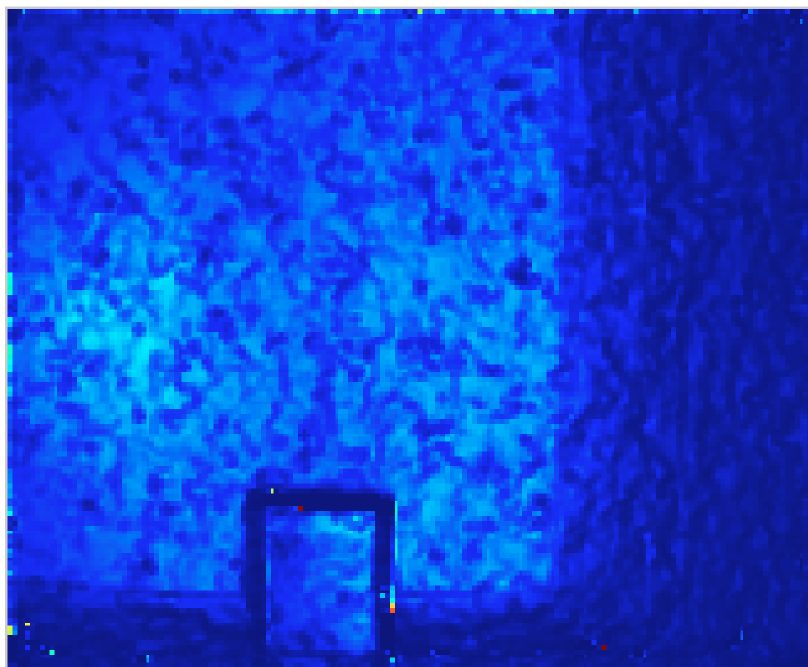

Figura 4.12: *Rho* de la imagen 2

Imagen 3

En esta tercera imagen no es posible diferenciar nada en la captura de amplitud.

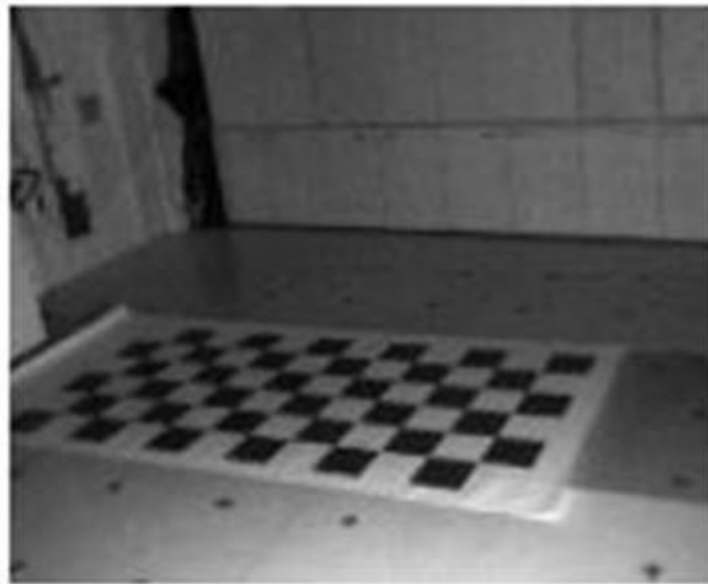

Figura 4.13: Amplitud de la imagen 3

Visualizando las normales es posible diferenciar dos paredes junto con el suelo de un entorno. Las normales de cada superficie se suponen correctas, siendo las del suelo orientadas verticalmente y las de las paredes perpendiculares a ellas.

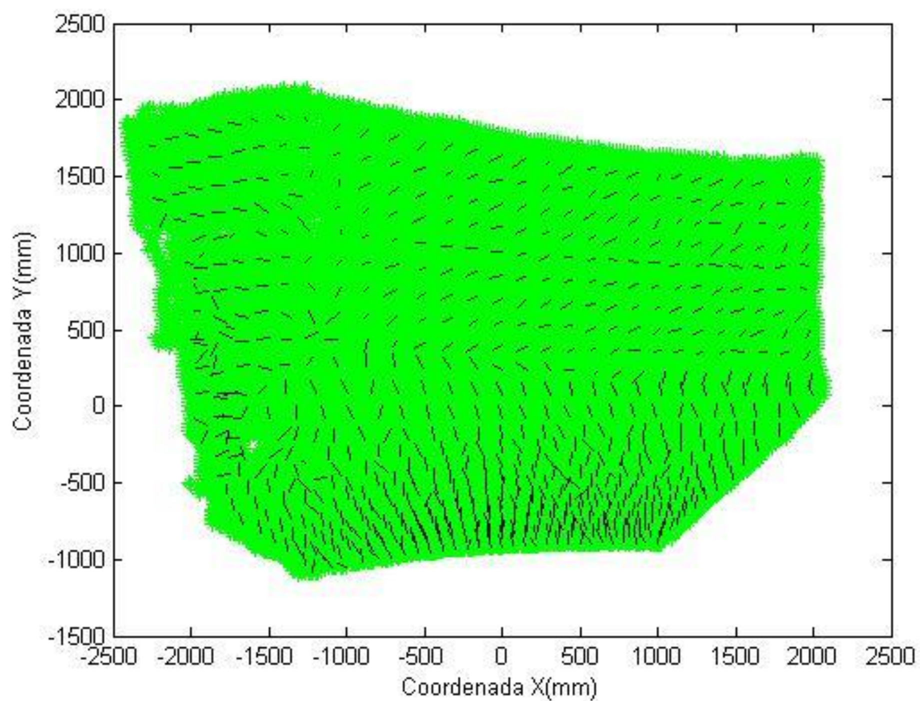

Figura 4.14: Normales de la imagen 3

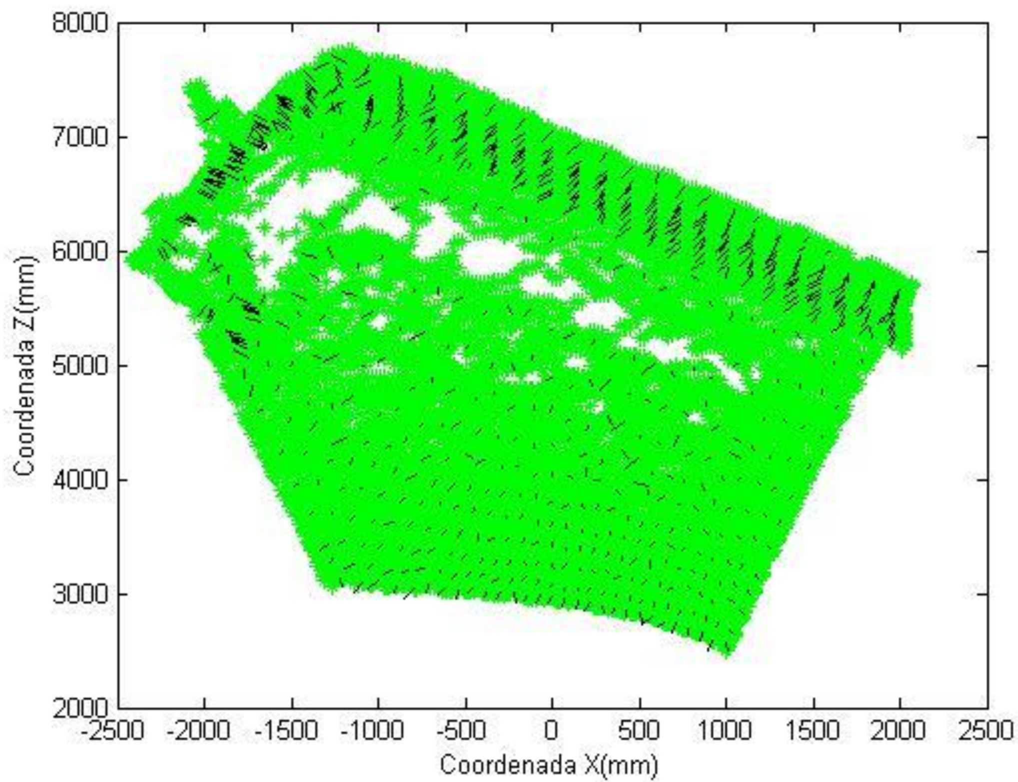

Figura 4.15: Normales de la imagen 3

Mediante la imagen del diferencial de área se corrobora lo anterior, viéndose el suelo y ambas paredes.

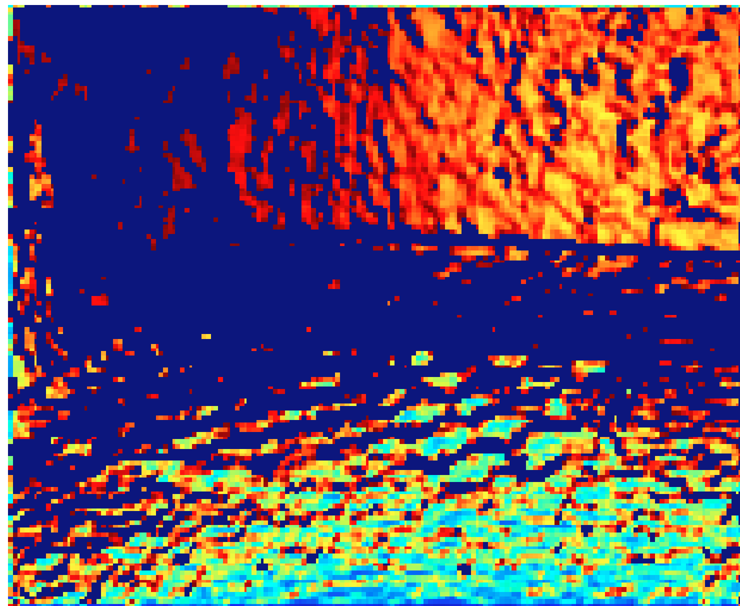

Figura 4.16: Diferencia de área de la imagen 3

Finalmente con la visualización de los datos obtenidos de *rho*, la imagen cobra más claridad, diferenciándose en ella un patrón de ajedrez en el suelo de la estancia.

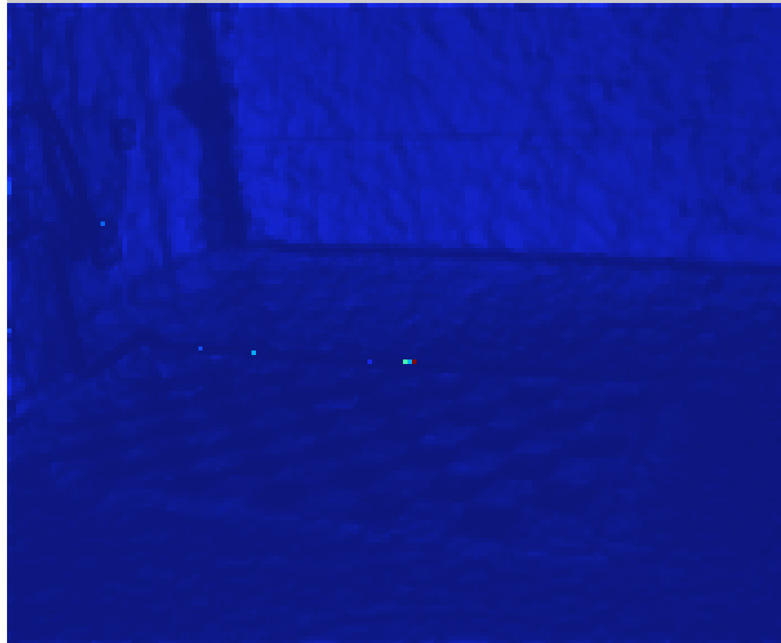

Figura 4.17: *Rho* de la imagen 3

4.2. Análisis temporal

En este apartado se presentan los tiempos de cómputo de las diferentes funciones implementadas en CUDA, así como una comparación entre los tiempos obtenidos en este trabajo, y los presentados en [8] para la implementación en Matlab y C.

Todos los tiempos mostrados se han obtenido al ejecutar cada una de las funciones en un ordenador portátil cuyas características hardware y software se detallan en el Anexo 1.

4.2.1. Tiempos medios de cada una de las funciones

En este apartado se realiza un estudio de los tiempos obtenidos para cada una de las funciones para veinticuatro imágenes diferentes de entrada, pudiendo obtenerse la media, desviación estándar y varianza de todos ellos.

Tiempo en el cálculo de la normal

Se realiza el estudio de tiempos para ambos casos del cálculo de normales, obteniéndose para radio de 1 píxel y de 2 píxel.

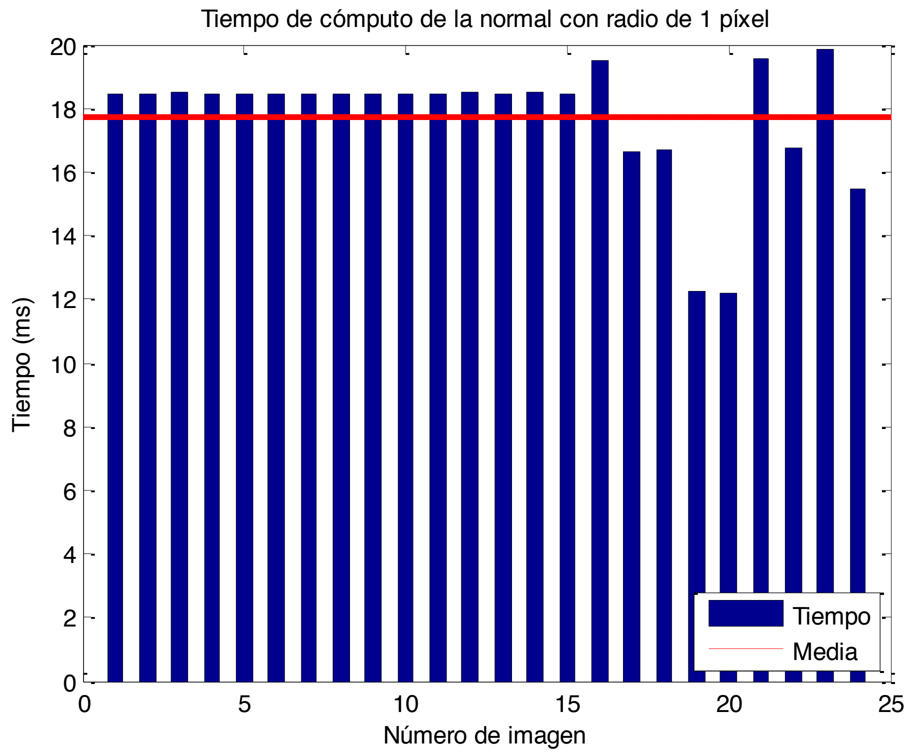

Figura 4.18: Tiempo de cómputo de la normal con radio de 1 píxel

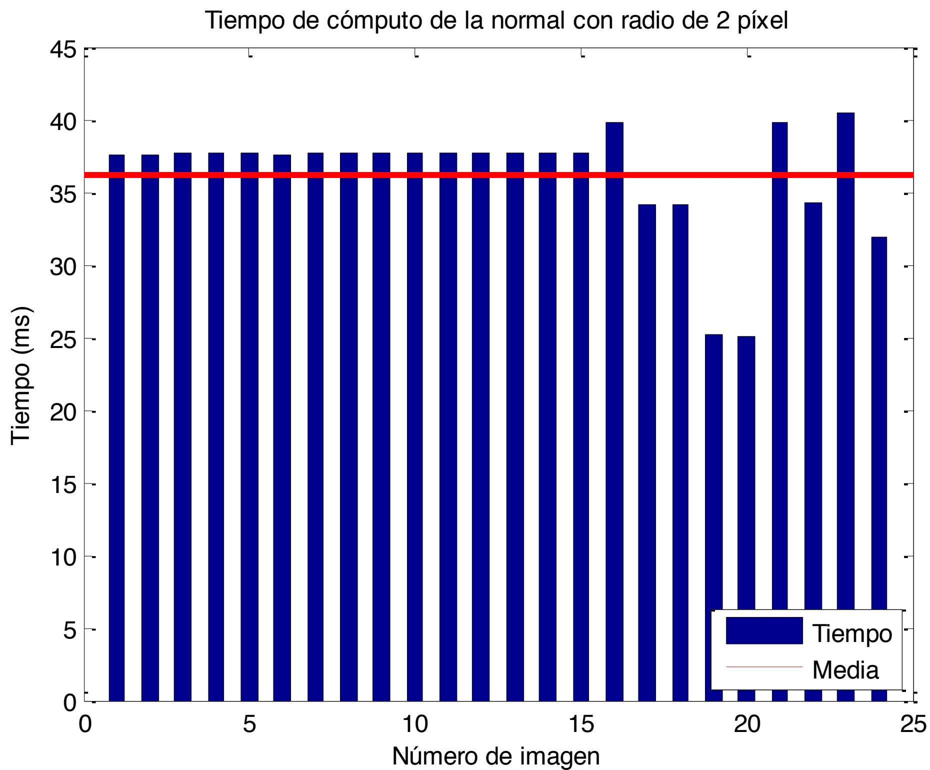

Figura 4.19: Tiempo de cómputo de la normal con radio de 2 píxel

Simplemente observando las gráficas se puede deducir que si para el cálculo de las normales se utiliza radio de 1 píxel, el tiempo de cómputo se asemeja más a tiempos reales. Esto conlleva a su vez que los valores de normal obtenidos sean menos precisos que en el caso de usar 2 píxel de radio.

La Tabla 4.1 muestra los resultados para los tiempos de esta función en ambos casos.

Función	Media	Desviación estándar	Varianza
Normal radio 1	17,75 ms	1,96 ms	3,85 ms^2
Normal radio 2	36,25 ms	3,89 ms	15,18 ms^2

Tabla 4.1: Variaciones temporales de la función normal

Tiempo en el cálculo del diferencial de área

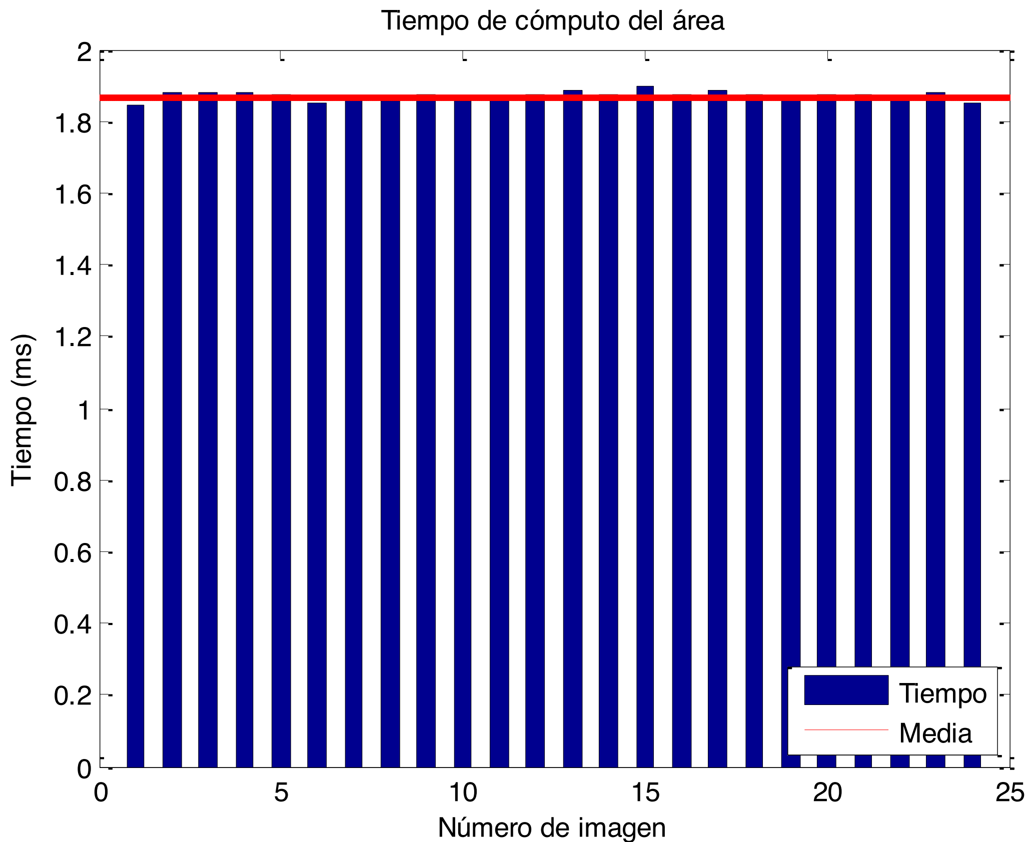

Figura 4.20: Tiempo de cómputo del diferencial de área

Debido a que el diferencial de área debe calcularse en todos los píxeles y no depende de otros factores (como puede ser el número de vecinos), los tiempos de cómputo son todos muy semejantes.

Función	Media	Desviación estándar	Varianza
Diferencia de área	1,87 ms	0,0127 ms	1,6e-4 ms^2

Tabla 4.2: Variaciones temporales de la función diferencial de área

Tiempo en el cálculo de rho

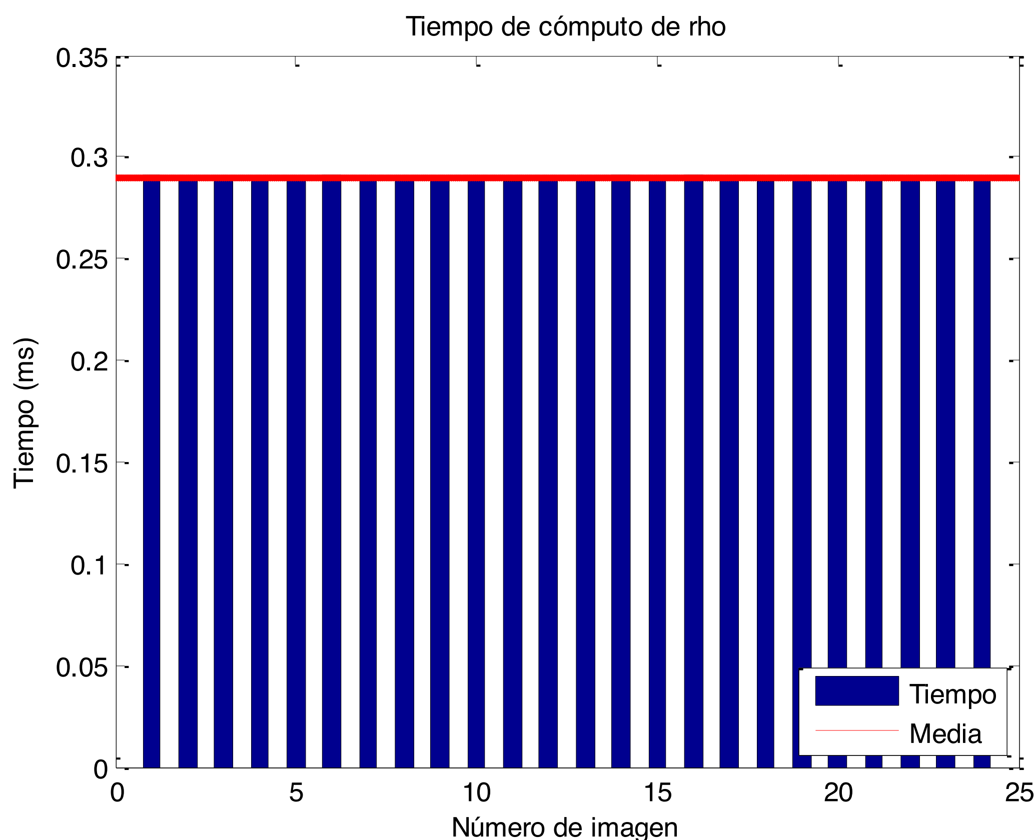

Figura 4.21: Tiempo de cómputo de *rho*

Igual que para el cálculo del diferencial de áreas, esta función debe realizarse para todos los píxeles indistintamente, por lo que también los tiempos son muy similares para todas las imágenes procesadas.

Función	Media	Desviación estándar	Varianza
Rho	0,2898 ms	9.32e-4 ms	8,7e-7 ms^2

Tabla 4.3: Variaciones temporales de la función *rho*

Conclusiones

La Figura 4.22 muestra el valor medio y la desviación típica de los tiempos de cómputo de cada una de las funciones para las 24 imágenes.

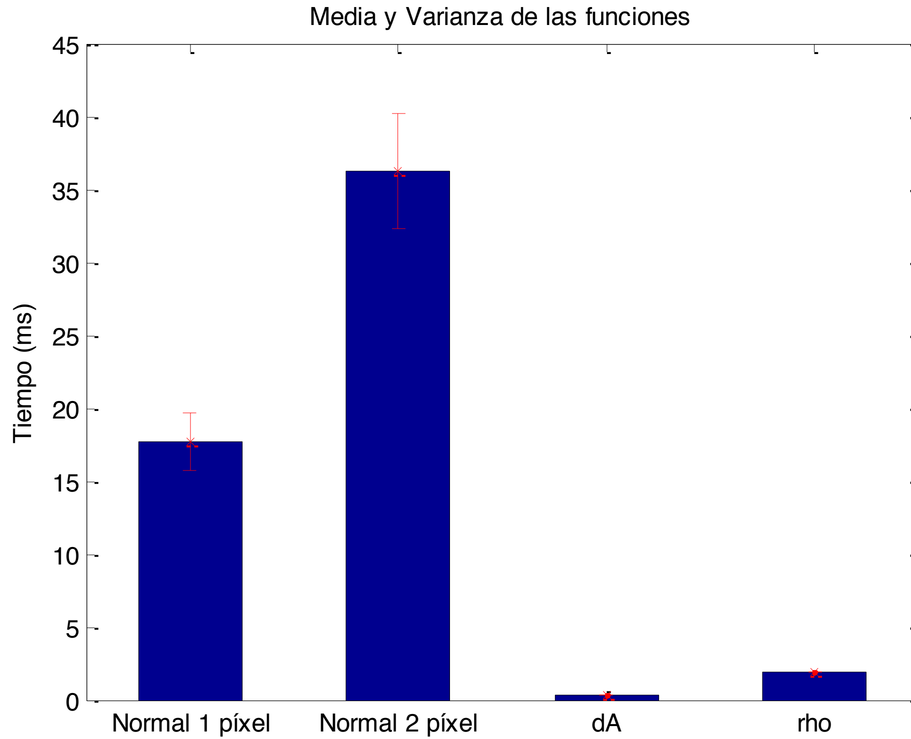

Figura 4.22: Valor medio y desviación típica del tiempo de cómputo de cada una de las funciones

Comparando todos los resultados, la función de la normal es la que mayor desviación estándar presenta, esto se debe a que el cálculo de los vecinos es diferente para cada imagen, variando el número de estos para cada píxel.

En el caso del resto de funciones las desviaciones son mínimas, incluso despreciables, debido a que debe realizarse para todos los píxeles por igual.

La Tabla 4.4 muestra la suma de los tiempos medios de todas las funciones para ambos casos del cálculo de la normal

Función	Sumatorio de tiempos
Normal radio 1+*dA*+*Rho*	19,91 ms
Normal radio 2+*dA*+*Rho*	38,41 ms

Tabla 4.4: Sumatorio de tiempos

A la luz de estos resultados, utilizar la función de normal con radio 1 se asemeja más a trabajar en tiempo real, pero como ya se comentó, la precisión de las normales obtenidas es menor que en el caso de radio 2.

4.2.2. Comparativa con otros entornos de programación

La Figura 4.23 muestra una comparativa de los tiempos de cómputo del algoritmo de corrección del *MpI* programado en distintos entornos de programación. En concreto se comparan los tiempos para las funciones programadas en Matlab, C y CUDA.

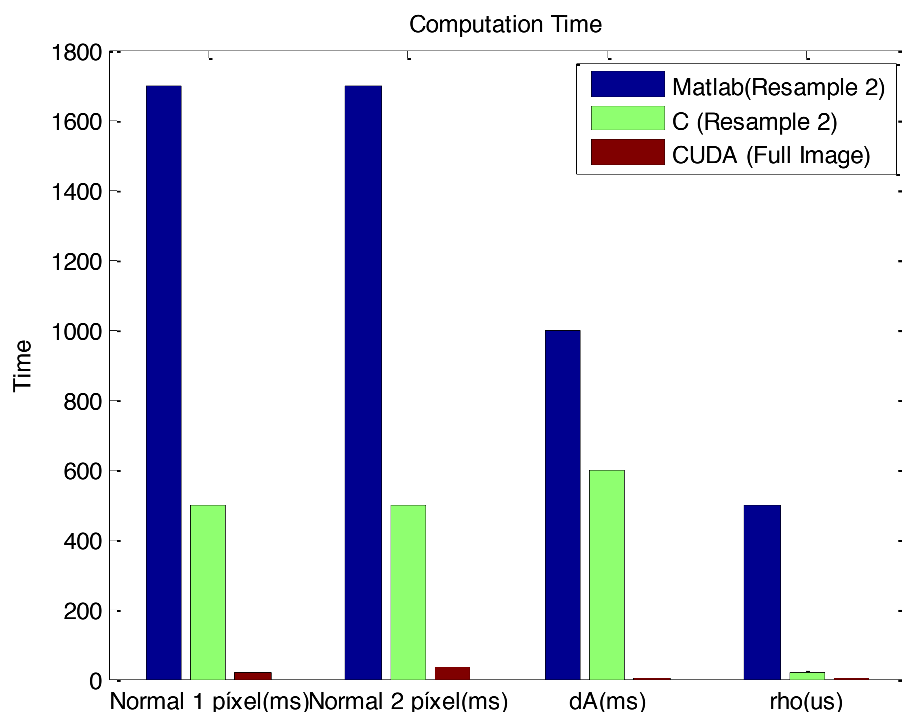

Figura 4.23: Comparativa de tiempos

El objetivo principal del trabajo era reducir los tiempos de cómputo de las funciones programadas en C. Viendo la Figura 4.23, se comprueba que los tiempos han sido reducidos, aun teniendo en cuenta que los mostrados en los otros entornos son tiempos para un *resample* de 2 y el obtenido con CUDA emplea la imagen completa.

Estos resultados nos permiten concluir que mediante programación en paralelo los tiempos de cómputo para funciones iguales se reducen considerablemente.

Capítulo 5.Conclusiones y trabajo futuro

Concluido el desarrollo del trabajo, se consideran cumplidos los objetivos propuestos en el inicio.

Se han programado las funciones con mayor coste computacional del algoritmo de corrección del *MpI* propuesto en [22] para aprovechar la capacidad de cálculo en paralelo de la GPU. Para ello, entre las diferentes alternativas se ha decidido utilizar CUDA.

Los resultados conseguidos para las normales, el diferencial de área y los *rho* relativos mediante la aplicación del algoritmo se consideran aptos, como se mostró en los resultados con diversas imágenes. A su vez hay que recordar que en todo momento el algoritmo trabaja con imágenes completas, siendo innecesarios realizar *resamples* como en [8] y en [22], consiguiendo así una mayor precisión en los resultados.

Respecto al tiempo de procesamiento de las funciones modificadas, tras paralelizar los algoritmos se ha logrado una reducción notable respecto a las versiones previas programadas en entornos C y/o Matlab [8], haciendo apto el sistema para trabajos en tiempo real. Cabe destacar en este punto la disyuntiva que existe entre precisión y velocidad al elegir el radio de vecindad del cálculo de las normales, consiguiendo mejores tiempos de cómputo (más próximos al tiempo real) para radios pequeños.

En definitiva, trabajar con programación paralela permite disminuir el tiempo de cómputo de las funciones más costosas programadas en C, pudiendo asumir que el sistema puede llegar a ser utilizado en aplicaciones de tiempo real.

Este trabajo abre una línea de trabajo para la implementación en GPU de algoritmos de visión computacional de cara al procesamiento en tiempo real, existiendo diferentes líneas de trabajo futuro.

Por un lado, se plantea como trabajo futuro la revisión y posible implementación del resto de funciones del algoritmo de corrección de *MpI* en GPU, ya que en el presente trabajo sólo se paralelizan tres de ellas.

También sería recomendable probar el presente algoritmo con otro tipo de tarjetas NVIDIA con una capacidad de cómputo mayor, pudiendo conseguir así tiempos aún más reducidos.

Anexo 1. Características del sistema

Características del Procesador

Atributo	Valor
Name	Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
Architecture	x64
Frequency	3.392 MHz
Number of Cores	8
Pages size	4.096
Total Physical Memory	8.175,00 MB

Características del Sistema Operativo

Atributo	Valor
Version Name	Windows 7 Professional
Version Number	6.1.7601

Características NVIDIA GeForce GT 630

Atributo	Valor
Driver Version	332.88
Driver Model	WDDM
CUDA Device Index	0
GPU Family	GF108
Compute Capability	2.1
Number of SM	2
Frame Buffer Physical Size (MB)	2048
Frame Buffer Bandwidth (GB/s)	25,6
Frame Buffer Bus Width (bits)	128
Frame Buffer Location	Dedicated
Graphics Clock (MHz)	700
Memory Clock (MHz)	800
Processor Clock (MHz)	1400
RAM Type	DDR3

Características para capacidad de cómputo de 2.1

Límites físicos de la GPU para capacidad de computo	2.1
Select Shared Memory Size Config (bytes)	49152
Threads per Warp	32
Warps per Multiprocessor	48
Threads per Multiprocessor	1536
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	32768
Register allocation unit size	64
Register allocation granularity	warp
Registers per Thread	63
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	128
Warp allocation granularity	2
Maximum Thread Block Size	1024

Bibliografía

[1] S. Lee, "Depth camera image processing and applications," in 19th IEEE International Conference on Image Processing, 2012, pp. 545–548.

[2] R. A. El-iaithy, J. Huang, and M. Yeh, "Study on the Use of Microsoft Kinect for Robotics Applications," in 2012 IEEE/ION Position Location and Navigation Symposium, 2012, pp. 1280–1288.

[3] P. Gemeiner, P. Jojic, and M. Vincze, "Selecting good corners for structure and motion recovery using a time-of-flight camera," in IEEE/RSJ International Conference on Intelligent Robots and Systems, 2009, pp. 5711–5716.

[4] J. R. Ruiz-Sarmiento, C. Galindo, and J. Gonzalez, "Improving Human Face Detection through TOF Cameras for Ambient Intelligence Applications," Appl. Adv. Intell. Soft Comput. Ambient Intell. - Softw. Appl., vol. 92, pp. 125–132, 2011.

[5] "Microsoft Kinect." [Online]. Available: http://www.xbox.com/en-us/kinect (accesible en junio 2014).

[6] "Asus XtionPro." [Online]. Available: http://www.asus.com/Multimedia/Xtion_PRO/ (accessible en junio 2014).

[7] R. Lange and P. Seitz, "Solid-state time-of-flight range camera," IEEE J. Quantum Electron., vol. 37, no. 3, pp. 390–397, 2001.

[8] Francisco Pérez Fermoselle (2012). Implementación de un Algoritmo para la Corrección de la Interferencia por Multicamino en Cámaras de Tiempo de Vuelo.

[9] Chiabrando, F., Piatta, D., and Rinaudo, F. (2010). SR4000 ToF Camera: Further Experimental Tests And First Applications To Metric Surveys. In ISPRS Commission V, Mid-Term Symposium, Close Range Image Measurement Techniques. ISPRS.

[10] Rapp, H., Frank, M., Hamprecht, F., and Jahne, B. (2008). A theoretical and experimental investigation of the systematic errors and statistical uncertainties of Time-Of-Flight-cameras. International Journal of Intelligent Systems Technologies and Applications, 5(3):402–413.

[11] Oggier, T., Lehmann, M., Kaufmann, R., Schweizer, M., Richter, M., Metzler, P., Lang, G., Lustenberger, F., and Blanc, N. (2004). An all-solid-state optical range camera for 3D real-time imaging with sub-centimeter depth resolution (SwissRanger). In Proc. SPIE, volume 5249, pages 534–545.

[12] Oprisescu, S., Falie, D., Ciuc, M., and Buzuloiu, V. (2007). Measurements with ToF Cameras and Their Necessary Corrections. In International Symposium on Signals, Circuits and Systems, 2007. ISSCS 2007, volume 1.

[13] Falie, D. (2008). 3D image correction for time of flight (ToF) cameras. In Int. Conf. of Optical Instrument and Technology, pages 7156–133.

[14] Mure-Dubois, J. and Hügli, H. (2007). Real-time scattering compensation for time-of-flight camera. In Proc. of the ICVS.

[15] Lindner, M. and Kolb, A. (2009). Compensation of Motion Artifacts for Timeof- Flight Cameras. Dynamic 3D Imaging, pages 16–27.

[16] Guomundsson, S., Aanaes, H., and Larsen, R. (2007). Environmental effects on measurement uncertainties of time-of-flight cameras. In Signals, Circuits and Systems, 2007. ISSCS 2007. International Symposium on, volume 1, pages 1–4. IEEE.

[17] Falie, D. and Buzuloiu, V. (2008a). Distance errors correction for the time of flight (ToF) cameras. In Imaging Systems and Techniques, 2008. IST 2008. IEEE International Workshop on, pages 123–126. IEEE.

[18] Falie, D. and Buzuloiu, V. (2008b). Further investigations on ToF cameras distance errors and their corrections. In Circuits and Systems for Communications, 2008. ECCSC 2008. 4th European Conference on, pages 197–200. IEEE.

[19] May, S., Fuchs, S., Droeschel, D., Holz, D., and Nüchter, A. (2009b). Robust 3Dmapping with time-of-flight cameras. In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).

[20] Fuchs, S. (2010). Multipath interference compensation in time-of-flight camera images. Pattern Recognition, International Conference on, 0:3583–3586.

[21] Forsyth, D. and Ponce, J. (2002). Computer vision: a modern approach. Prentice Hall Professional Technical Reference.

[22] Jimenez, D., Pizarro, D., Mazo, M., and Palazuelos, S. (2012). Modelling and correction of multipath interference in time of flight cameras. In Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on. IEEE.