Instituto Universitario de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería

TRABAJO FIN DE MÁSTER

PARALELIZACIÓN DE UN ALGORITMO DE RAY TRACING PARA ARRAYS DE MILILENTES

Titulación: Máster Oficial en Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería
Autor: D. Antonio Cristo Suárez Rodríguez
Tutores: Dr. Eduardo M. Rodríguez Barrera
Dr. Domingo J. Benítez Díaz
Fecha: Julio de 2015

"There are three kinds of lies: lies, damned lies, and benchmarks."
Variación de una frase falsamente atribuida a Benjamin Disraeli por Mark Twain.

Índice general

Índice de figuras

Índice de tablas

Resumen

El uso de *arrays* de microlentes está ampliamente extendido en sistemas ópticos adaptativos como los sensores de frente de onda Shack-Hartmann, así como para homogeneizar patrones de radiación de diversos sistemas de iluminación. Asimismo, se emplean para fotografía plenóptica donde, a partir de una única toma, es posible obtener imágenes desde distintos ángulos, recrear la paralaje u obtener mapas de distancia. Estas imágenes, a su vez, pueden reproducirse de forma tridimensional empleando un *array* de mililentes que permite obtener un número finito de puntos de vista. Partiendo de esta última idea, situar un *array* de mililentes sobre una pantalla convencional permite que el conjunto se comporte como un modulador espacial de luz y propicia la aparición tanto de técnicas de conformado de haz como de división espacial.

En este Trabajo Fin de Máster se propone implementar un simulador que recoja este escenario y paralelizarlo sobre las arquitecturas paralelas de memoria compartida y de tipo procesador gráfico. Por este motivo, se ha implementado un trazador de rayos que obtenga las relaciones existentes entre la luz emitida por la fuente y la luz refractada por las mililentes. De esta forma, en etapas posteriores, será posible generar un funcional que describa dicha relación para poder ser usada en un proceso de síntesis de patrones (*beamforming*). Para maximizar los resultados obtenidos para el tipo de mililentes considerado, se ha decidido realizar un muestreo selectivo del espacio que obvie las regiones del espacio donde la luz no es capaz de propagarse.

Abstract

Microlens arrays are widely used in adaptive optics such as Shack-Hartmann wavefront sensors, as well as in illumination by making homogeneous radiation patterns. Besides, plenoptic cameras employ them to achieve, from just a single photograph, a range of images with different points of view which is capable of showing parallax or generating depth maps. These images can also be showed in a three-dimensional way by means of a millilens array that let us obtain a finite set of points of view. Keeping this in mind, placing a millilens array over a common display can be seen as a spatial light modulator and promotes the advent of beamforming and spatial-division techniques.

In this work, lenticular arrays are supposed to beam steering light from a display. Sequential simulation software has been parallelized in shared memory (OpenMP) and graphics processing unit (CUDA) for speed-up purposes. A ray tracer has been programmed to get the relationship between emitted light and refracted light by the millilenses. Thanks to it, in later steps, it will be possible to calculate a functional that will describe that relation to be used in a synthesis process. So as to maximize results regarding the considered millilens array, it is been designed a spatial-selective sampling that ignores regions where light cannot propagate itself.

Lista de acrónimos

Acrónimo	Descripción
ACO-OFDM	*Asymmetrically Clipped Optical* OFDM
ADC	*Analog-to-Digital Converter*
API	*Application Programming Interface*
CAFADIS	CÁmara FAse DIStancia
CPU	*Central Processing Unit*
CSK	*Color-Shift Keying*
CUDA	*Compute Unified Device Architecture*
DAC	*Digital-to-Analog Converter*
DCO-OFDM	DC-*Biased Optical* OFDM
DMA	*Direct Memory Access*
DTFC	División de Tecnología Fotónica y Comunicaciones
FFT	*Fast Fourier Transform*
FPGA	*Field Programmable Gate Array*
FSK	*Frequency-Shift Keying*
GPU	*Graphics Processing Unit*
IAC	Instituto Astrofísico de Canarias
IDE	*Integrated Development Environment*
IDeTIC	Instituto para el Desarrollo Tecnológico y la Innovación en Comunicaciones
IEEE	*Institute of Electrical and Electronics Engineers*
IFFT	*Inverse* FFT
IM/DD	*Intensity Modulation with Direct Detection*
MPI	*Message Passing Interface*
MSIANI	Máster Oficial en Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería
OFDM	*Orthogonal Frequency-Division Multiplexing*
OLED	*Organic Light-Emitting Diode*
OpenMP	*Open Multi-Processing*

Acrónimo	Descripción
PHY	*Physical Layer*
PN	*Positive-Negative*
PWM	*Pulse-Width Modulation*
RAM	*Random-Access Memory*
SDMA	*Spatial-Division Multiple Access*
SIMT	*Single Instruction, Multiple Threads*
SLM	*Spatial Light Modulator*
SM	*Streaming Multiprocessors*
TFM	Trabajo Fin de Máster
ULL	Universidad de La Laguna
VLC	*Visible Light Communications*
WDM	*Wavelength-Division Multiplexing*
WSN	*Wireless Sensor Networks*

Parte I

Memoria

Capítulo 1

Introducción

En el presente capítulo se expone la motivación del trabajo y se introduce una visión general del problema así como los objetivos considerados. Por último, se expone la distribución de los contenidos de esta memoria.

1.1. Motivación

Para entender cuál es la motivación de este trabajo, es necesario citar al grupo CÁmara FAse DIStancia (CAFADIS) de la Universidad de La Laguna (ULL). En colaboración con el Instituto Astrofísico de Canarias (IAC), se ha desarrollado un prototipo que detecta el frente de onda óptico y estima la distancia [1]. Dicho prototipo consiste en un conjunto de lentes y microlentes que puede adosarse a cualquier cámara convencional, desde cámaras réflex digitales hasta cámaras de cine.

Entre sus aplicaciones, en el año 2010 desarrollaron un algoritmo de 3DTV en tiempo real que alcanza hasta 200 puntos de vista en 24 planos focales distintos. En astronomía, su aplicación directa pasa por corregir las aberraciones producidas por las distintas capas de la atmósfera, produciendo imágenes de mayor calidad. Además, han implementado modelos tanto en software (en GPU, *Graphics Processing Unit*) como en hardware (en FPGA, *Field Programmable Gate Array*).

Desde el punto de vista de las comunicaciones, fruto de la colaboración entre el IDeTIC (Instituto para el Desarrollo Tecnológico y la Innovación en Comunicaciones) y el CAFADIS, en [2] se propone el uso de cámaras plenópticas en redes de sensores ópticas no guiadas. De forma inherente, emplear una cámara convencional como receptor introduce el concepto de diversidad espacial en el receptor, sin embargo, en este trabajo se plantea la utilización de plenópticas en escenarios con alta densidad de sensores donde se pueden dar casos de problemas *near-far*. Actualmente, dicha línea de investigación sigue activa a la espera de obtener resultados.

A raíz de esta última colaboración, surgió la posibilidad de emplear un *array* de mililentes en el lado transmisor con el fin de introducir técnicas de *Spatial-Division Multiple Access* (SDMA) en redes de sensores ópticas inalámbricas. Esta configuración

es precisamente la que se emplea en imagen integral donde, al situar un *array* de mililentes sobre una pantalla convencional, se consigue reproducir una imagen tridimensional sin emplear gafas u otros dispositivos. A este tipo de sistemas que no hacen uso de elementos externos se los conoce como sistemas autoesteroscópicos, en los que se puede recrear efectos de profundidad y/o paralaje.

No obstante, en nuestro caso, y centrándonos en las redes de sensores ópticas inalámbricas, no existe la necesidad de formar imágenes puesto que los canales son modulados en intensidad con detección directa (IM/DD, *Intensity Modulation with Direct Detection*). Por lo tanto, se trata de un subproblema de imagen integral adaptado a comunicaciones. Es aquí donde comienza este trabajo, intentando modelar dicho problema para adaptarlo a nuestras necesidades.

1.2. Descripción del problema

El título de este Trabajo Fin de Máster (TFM) hace alusión a la paralelización de un algoritmo, en concreto, al empleo de trazado de rayos para *arrays* de mililentes. Sin embargo, una de las primeras tareas a realizar para modelar el problema fue describir el escenario y los actores principales del mismo. Para ello, se parte del escenario mostrado en la Figura 1.1 donde se encuentran tres elementos: un *display*, un *array* de mililentes y un plano imagen.

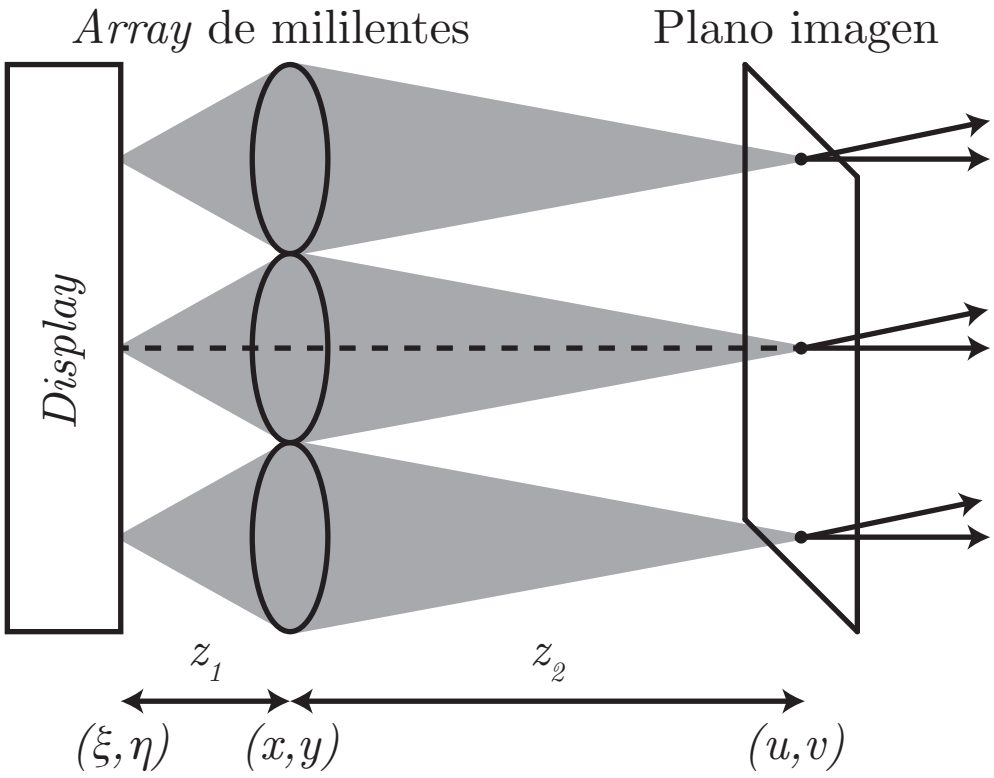

Figura 1.1: Escenario del problema

Por un lado, el *display* es el elemento emisor de información que se ha decidido modelar como un *array* de ledes; por otro lado, las mililentes se ha decidido que

aumenten un orden de magnitud respecto a las microlentes empleadas en una cámara plenóptica (de cientos de micras a unidades de milímetros, esto se discutirá en la Sección 3.1); y por último, el plano imagen que aunque no se corresponde con un elemento físico nos permite obtener resultados.

No se ha encontrado ningún trabajo en la literatura que aborde el estudio de este escenario con fines de comunicaciones. Sí se ha encontrado trabajos relacionados con la iluminación que sintetizan patrones sobre un *array* de ledes como hizo Moreno en [3] o que homogeneízan el patrón de radiación de un led por medio de *arrays* de mililentes tal como recoge Schreiber [4].

Por consiguiente, antes del planteamiento de este TFM, se pensó en obtener de forma analítica las ecuaciones que rigen el escenario presentado en la Figura 1.1 con el fin de obtener un conjunto de funciones base con las que realizar un proceso de síntesis. Pronto se comprobó que dicha tarea entrañaba una dificultad excesiva y se replanteó el problema para obtener mediante simulación el funcional descrito en la Ecuación 1.1.

$$\mathcal{S}\left(\mathcal{R}\left(\theta,\phi\right)\right)\ \text{W/sr} \tag{1.1}$$

donde $\mathcal{S}$ representa la intensidad radiante (W/sr) de una mililente en función de la intensidad radiante de un led, $\mathcal{R}$, que, a su vez, depende de las coordenadas esféricas θ y ϕ. Si planteamos la ecuación que describe el escenario completo de la Figura 1.1, se obtiene la Ecuación 1.2.

$$\mathcal{T} = \sum_i \sum_j \mathcal{S}_{i,j} \left(\sum_u \sum_v \alpha_{u,v} \mathcal{R}_{u,v}\left(\theta_{i,j}, \phi_{i,j}\right) \right)\ \text{W/sr} \tag{1.2}$$

en este caso, se ha generalizado la Ecuación 1.1 donde la intensidad radiante de una mililente viene conformada por la intensdiad radiante de muchas fuentes moduladas en intensidad por el parámetro α.

Es por ello que para obtener dicho funcional se pensó en un simulador que recogiese un número significativo de casos, para posteriormente realizar un ajuste de funciones para un escenario dado. Planteando *grosso modo* el sistema a optimizar, este consiste en: un primer bloque de simulación, una obtención del funcional para ese escenario y una extracción de parámetros. El criterio de parada para la síntesis es un determinado patrón de radiación a una distancia dada, de ahí el plano imagen. El flujo del sistema se presenta en la Figura 1.2.

Carece de sentido modelar el sistema desde el punto de vista de la óptica física puesto que no pueden sucederse fenómenos de difracción ni de interferencia debido a la diferencia relativa de dimensiones entre la longitud de onda (nanómetros) y el sistema óptico (milímetros). Además, el uso de fuentes no coherentes como los ledes hace que sólo se pueda trabajar en intensidad. La opción lógica pasa por modelar de forma geométrica.

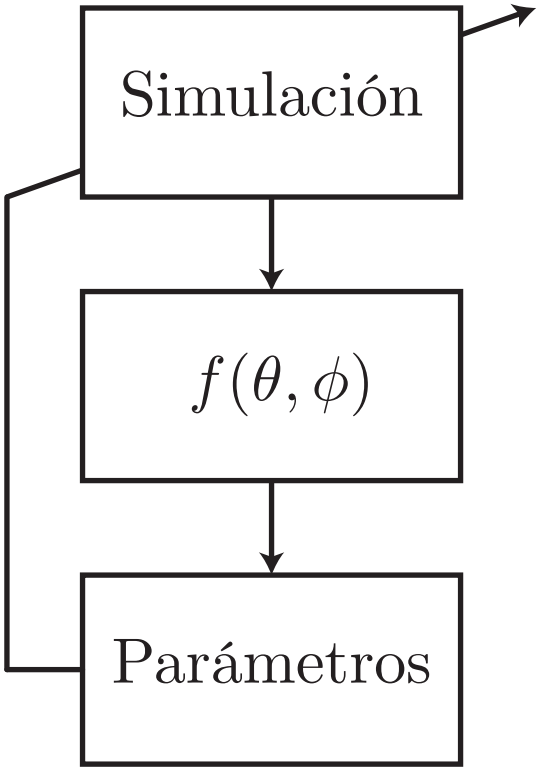

Figura 1.2: Flujo del sistema

1.3. Objetivos

Este TFM tiene como objetivo principal la implementación utilizando las arquitecturas paralelas de memoria compartida y de tipo GPU de una aplicación de comunicaciones ópticas basada en técnicas de *ray tracing*. A su vez, este objetivo se desglosa en los siguientes:

- Estudiar la aplicación de las diferentes arquitecturas de paralelización sobre el problema de *ray tracing*.
- Análisis y diseño de la aplicación de *ray tracing* en las arquitecturas paralelas de memoria compartida y de tipo GPU.
- Implementación del código en las arquitecturas paralelas de memoria compartida y de tipo GPU.
- Evaluación de las prestaciones de la aplicación de *ray tracing* en las arquitecturas paralelas elegidas.

1.4. Organización de la memoria

El presente trabajo se ha dividido de la siguiente manera:

- **Parte I Memoria**
  Se presenta la memoria del trabajo, compuesta de seis capítulos detallados a continuación:

  - **Capítulo 1 Introducción**
    Se describen aspectos generales del problema a abordar, los objetivos del presente TFM, así como la estructura del documento.
  - **Capítulo 2 Antecedentes**
    Se realiza una descripción del estado del arte de las distintas disciplinas involucradas.
  - **Capítulo 3 Modelo matemático**
    Se define el modelo matemático seguido así como el algoritmo resultante.
  - **Capítulo 4 Implementación**
    Se explica el proceso seguido para implementar el código en las distintas arquitecturas.
  - **Capítulo 5 Resultados**
    Se presenta y analiza los resultados obtenidos en distintas fases del trabajo.
  - **Capítulo 6 Conclusiones**
    Se detalla las conclusiones obtenidas a raíz de los resultados conseguidos.

- **Parte II Bibliografía**
  Se indica la bibliografía consultada para la realización del TFM.

Capítulo 2

Antecedentes

En este capítulo se esboza el estado del arte de aquellas disciplinas que entroncan con la realización de este trabajo. En concreto, se abordan temas de comunicaciones por luz visible, visores tridimensionales y tecnologías de paralelización.

2.1. Comunicaciones por luz visible

Dentro de las comunicaciones ópticas no guiadas, el uso del canal VLC (*Visible Light Communications*) como enlace de bajada no sólo ofrece un amplio ancho de banda sin regular y una alternativa a las bandas 2,4 GHz ya muy saturadas, sino que además usa una infraestructura ya existente en todas las viviendas u oficinas. A consecuencia de esto se ha producido la eclosión de distintas iniciativas normativas como el estándar IEEE (*Institute of Electrical and Electronics Engineers*) 802.15.7 [5]. Su PHY (*Physical Layer*) III denominada CSK (*Color-Shift Keying*) es similar a una FSK (*Frequency-Shift Keying*) aunque en longitud de onda, no confundir con la multiplexación por longitud de onda (WDM, *Wavelength-Division Multiplexing*) donde se empelan canales independientes. En concreto, la CSK se basa en la codificación de la información en las amplitudes relativas entre las portadoras ópticas [6]. Aunque en el estándar se presentan una serie de reglas para la generación de constelaciones, estas pueden cambiarse empleando algoritmos de optimización como en [7] [8] o para añadir capacidades multiusuario directamente en la capa física [9].

Aparte de las técnicas de codificación propuestas por el estándar, durante los últimos años multitud de grupos de investigación han estudiado el uso de otras modulaciones para implementar sistemas VLC [10]. Por ejemplo, se han explorado el uso de técnicas OFDM (*Orthogonal Frequency-Division Multiplexing*) con el fin de lograr sistemas de comunicaciones con una eficiencia espectral alta y elevada robustez frente a multipropagación [11]. Para poder emplear estas señales bipolares en el dominio óptico se recurre comúnmente a dos técnicas bien diferenciadas: DCO-OFDM (DC-*Biased Optical* OFDM) y ACO-OFDM (*Asymmetrically Clipped Optical* OFDM) [12]. La primera de ellas se basa en introducir un nivel de continua para polarizar al led en la región lineal, mientras que la segunda opción elimina las componentes pares de la IFFT (*Inverse* FFT), reduciendo a la mitad la eficiencia espectral de la modulación

tal y como se puede comprobar en la Figura 2.1. Recientemente se han combinado ambas técnicas aprovechando subportadoras alternas, en las pares se modula en DCO mientras que en las impares se emplea ACO; incrementando la complejidad del receptor al requerir de bloques de cancelación de ruido [12]. Para concluir, otra alternativa basada en una patente [13] serializa las partes positivas y negativas de la señal OFDM en dos *subframes*. Aunque no ha trascendido de forma relevante en la literatura, en [14] se concluye que es equivalente a una ACO, reduciendo la complejidad a la mitad.

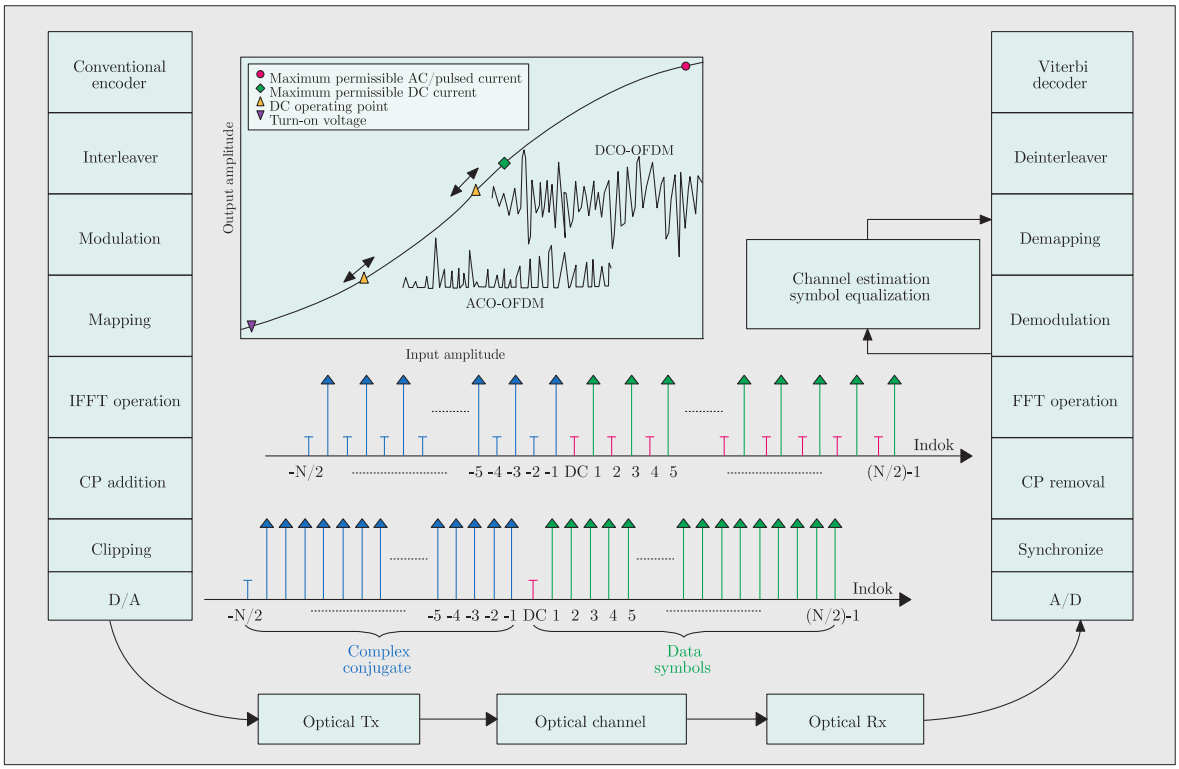

Figura 2.1: Esquemas DCO-OFDM y ACO-OFDM

A pesar de que ya se han realizado implementaciones en dispositivos programables como FPGA, estos sistemas conllevan un alto coste en complejidad dada la necesidad de realizar las FFT (*Fast Fourier Transform*) e IFFT, y requieren el uso de ADC (*Analog-to-Digital Converter*) y DAC (*Digital-to-Analog Converter*) capaces de trabajar a elevadas frecuencias de muestreo. Además, es necesario emplear dispositivos completamente lineales en todo el ancho de banda de trabajo, tanto en la cadena transmisora como en la receptora. El uso de un amplificador Clase D como modulador PWM (*Pulse-Width Modulation*) puede aumentar la eficiencia energética, entendida como eficiencia de conversión optoeléctrica, como se muestra en [15]. Por último, en [16] se hace uso de un SLM (*Spatial Light Modulator*) en amplitud y fase para enfocar el haz en un punto concreto. El uso de un array de microlentes podría salvar esta limitación y concentrar la radiación en diferentes puntos en lugar de uno único.

2.2. Visores tridimensionales

Como se comentó en la Sección 1.1, puede emplearse un *array* de mililentes para formar imágenes tridimensionales, formando lo que se conoce como sistema autoestereoscópico donde no se requiere del uso de gafas. Este efecto también se puede conseguir mediante técnicas de *eye-tracking* pero no se consideran en este trabajo porque no se necesita formar imágenes para un usuario humano, sino dentro de un contexto de comunicaciones.

Además, en VLC no se emplea fuentes coherentes de luz (láseres) y sus canales son IM/DD, lo cual descarta otras técnicas de visión tridimensional como puede ser la holografía [17]. No obstante, si el objetivo fuese la formación de imágenes hay que tener en cuenta las limitaciones de la estereoscopía donde dependiendo del ángulo de visión se pueden formar imágenes borrosas (*cross-talk* en forma de *ghosting*) como resultado de la combinación de dos puntos de vista. Con todo, en este trabajo se propone el uso de sistemas autoestereoscópicos para poder introducir técnicas de SDMA en WSN (*Wireless Sensor Networks*). Entre las más conocidas destacan la barrera de paralaje y las lentes lenticulares como se muestra en la Figura 2.2.

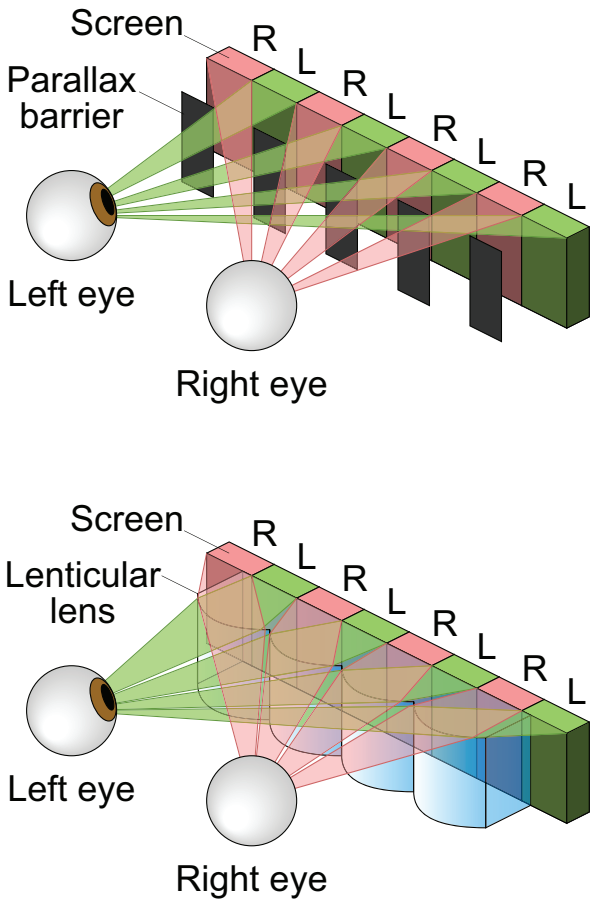

Figura 2.2: Comparación entre barrera de paralaje y lentes lenticulares

2.2.1. Barrera de paralaje

Inventada de forma independiente por Auguste Bertier (1896) y Frederic E. Ives (1901), esta técnica se basa en la introducción de una barrera compuesta de zonas opacas y transparentes de forma que, colocadas convenientemente sobre un visor, genera una imagen distinta para cada ojo. Esta barrera puede ser estática, con lo que el efecto tridimensional se mantiene, o dinámica por medio de un cristal líquido, donde se puede conmutar entre 2D y 3D. Esta última técnica se emplea en consolas de videojuegos como la Nintendo 3DS [18]. De forma intuitiva esta técnica permite únicamente un punto de vista tridimensional donde dos imágenes 2D son intercaladas para formar una única imagen 3D, con la consecuente pérdida de resolución del visor original.

2.2.2. Lentes lenticulares

De la misma forma que la técnica anterior, se entrelazan, al menos, dos imágenes y sobre la imagen resultante se alinea de forma precisa un *array* de lentes. Esta es la base sobre la que Gabriel Lippmann propuso en 1908 el concepto de imagen integral. La luz proveniente de la imagen que puede ser emitida o reflejada, según sea activa o pasiva la fuente, se refracta sobre la superficie de la lente en ángulos ligeramente diferentes. La luz de una imagen concreta se refracta siempre en la misma dirección. En función del número de imágenes empleado se puede obtener más de un punto de vista, como por ejemplo en [19] donde emplean 9. Por otro lado, el perfil de la lente planoconvexa que se utilice permite más o menos grados de libertad, por ejemplo, si las lentes son cilíndricas, la luz se refracta en forma de abanico.

2.3. Tecnologías de paralelización

Las tecnologías de paralelización elegidas sobre las que adaptar el simulador secuencial son tanto de memoria compartida (OpenMP, *Open Multi-Processing*) como de tipo GPU (**CUDA®**, *Compute Unified Device Architecture*). Existe otro tipo de tecnología basada en memoria distribuida (MPI, *Message Passing Interface*), sin embargo, a la hora de comparar tecnologías se descartó puesto que está orientada a las arquitecturas multicomputador.

2.3.1. Memoria compartida

OpenMP [20] es la API (*Application Programming Interface*) estándar *de facto* a la hora de escribir aplicaciones paralelas que empleen el paradigma de memoria compartida. Es una extensión a los lenguajes de programación C, C++ y Fortran; a los cuales añade una serie de directivas de compilación, bibliotecas y variables de entorno. La programación con OpenMP no requiere un gran esfuerzo. Se parte de un algoritmo paralelizable, se modifica a través de una serie de *pragmas* y se asegura que no se den condiciones de carrera por una mala estrategia de paralelización. Se basa en el modelo *fork/join* que se muestra en la Figura 2.3.

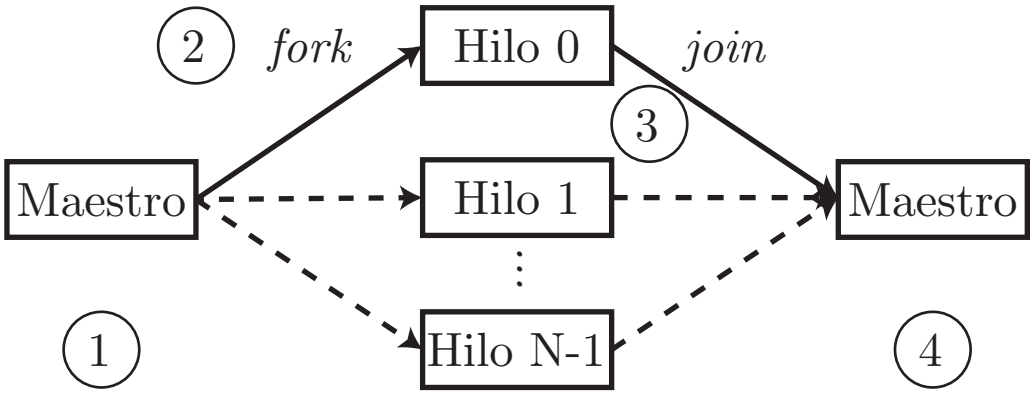

Figura 2.3: Esquema de funcionamiento del modelo *fork/join*

1. Un programa escrito en OpenMP comienza con un único hilo, denominado hilo maestro.

2. Al encontrar una sección de código paralelo, el hilo maestro crea un grupo de hilos auxiliares (*fork*).

3. Las sentencias dentro de la sección paralela se ejecutan de forma secuencial por cada hilo.

4. Al finalizar dicha sección, los hilos se sincronizan (barrera implícita) y se destruyen quedando sólo el maestro (*join*).

El formato básico de las directivas en C es el siguiente:

```
#pragma omp parallel [clauses]
```

De esta forma se define una región paralela, la cual va a ser ejecutada por múltiples hilos. Es la directiva fundamental que comienza la ejecución en paralelo. Dentro de las opciones disponibles para especificar cómo se repartirá la carga de trabajo tenemos:

- `for`: se divide un bucle según su número de iteraciones (n) entre el número de hilos (h). Junto con la cláusula `schedule [type, chunk]` permite cierto control sobre el reparto de las iteraciones entre los hilos:
  - `static`: cada hilo ejecuta `chunk` iteraciones antes de continuar.
  - `dynamic`: como `static` pero cada hilo no espera por la sincronización para continuar.
  - `guided`: parte de n/h iteraciones por hilo y decrece exponencialmente hasta un mínimo de `chunk`.
- `sections`: puede ejecutarse en paralelo pero sólo por un único hilo.
- `single`: como `sections` pero obliga al resto de hilos a sincronizarse.
- `master`: bloque que sólo puede ejecutar el hilo maestro, sin sincronización.

Por defecto, todas las variables que se emplean dentro de una sección de código paralelo son compartidas (`shared`) salvo que se especifique lo contrario mediante las directivas oportunas (`private`). Excepciones a esta regla son las variables índice de un bucle. Sin embargo, una vez acabada la región paralela, el valor de las variables privadas queda indefinido. Es por ello que existen las cláusulas:

- `lastprivate`: el valor de salida de la variable privada será el de la última iteración del código paralelo entendido en el sentido secuencial.
- `firstprivate`: indica que el estado inicial de la variable local es el que tenía justo antes del código paralelo.
- `reduction (op:list)`: realiza la reducción de variables escalares empleando el operador indicado.

De forma implícita, cada vez que se finaliza una sección de código paralelo, todos los hilos se sincronizan. Aún así, a veces es deseable por parte del programador decidir cuándo dicha sincronización es necesaria. OpenMP permite mediante las siguientes cláusulas especificarlo:

- `critical`: indica que sólo un hilo a la vez puede acceder a dicho bloque, de esta forma, se protege el valor de alguna variable compartida.
- `barrier`: sincroniza todos los hilos.
- `nowait`: hace que la barrera implícita sea ignorada.

Para establecer el número de hilos pueden utilizarse dos opciones:

- Variable de entorno: `OMP_NUM_THREADS`.
- Función: `omp_set_num_threads (int n)`.

2.3.2. GPU

Las GPU trasladan al sector de consumo la calidad visual de la síntesis de imagen tridimensional fotorrealista, esforzándose en permanecer en tiempos de ejecución interactiva. Las nuevas generaciones de tarjetas gráficas no sólo han elevado en varios órdenes de magnitud la potencia de computacional de las generaciones previas, superando con ello en prestaciones a las CPU (*Central Processing Unit*), sino que además presentan un modelo de programación más abierto. Este potencial, en un periférico de bajo coste y de uso extendido, ha llamado la atención de la comunidad científica, que se las ha ingeniado para realizar cómputos genéricos sobre una plataforma diseñada en principio para tareas específicas. Entre ellos el cómputo de transformadas discretas, el álgebra lineal, las ecuaciones diferenciales, las técnicas *multi-grid*, la

dinámica de fluidos, el crecimiento cristalino, la planificación de movimiento, el trazado de rayos, y un largo etcétera.

La diferencia en rendimiento respecto al de arquitecturas convencionales se logra haciendo predominar las unidades de cómputo frente a *caches* y lógica de control sobre la superficie del chip como se muestra en la Figura 2.4. Relajando la complejidad de la lógica de programa es posible aprovechar en mayor medida el paralelismo implícito en cierto tipo de problemas los cuales se pueden beneficiar del uso de cientos de *cores*.

Figura 2.4: Comparación del espacio dedicado a cómputo entre una CPU y una GPU

En 2006, Nvidia presentó **CUDA®** como una plataforma de cómputo de propósito general sobre sus propias GPU. Desde entonces, se ha convertido en un entorno de programación que permite usar de forma nativa una variación de los lenguajes de alto nivel C/C++ y Fortran. Por medio de *wrappers*: Python y Java, entre otros. La arquitectura de dicho entorno está compuesta por varios componentes, los cuales se representan gráficamente en la Figura 2.5.

A pesar de que Nvidia ya ha desarrollado una nueva arquitectura para sus GPU, Maxwell, la unidad de la que dispone la División de Tecnología Fotónica y Comunicaciones (DTFC) es una GeForce GTX 680 [21] con arquitectura Kepler. La Figura 2.6 muestra la arquitectura de las GPU Kepler. La memoria principal del *host* y la memoria principal de la GPU se comunican a través de DMA (*Direct Memory Access*), controlando las transacciones de memoria en el *host*. Los hilos son gestionados por la unidad de control de la GPU (*GigaThread Engine*).

Como Fermi, la generación anterior, la GPU está formada por grupos de procesadores que ejecutan las instrucciones al mismo tiempo. El gran cambio respecto a Fermi reside en aumentar el rendimiento por vatio a través de los nuevos *Streaming Multiprocessor* (SM). Los SMX son el núcleo de la arquitectura propuesta por Nvidia. Por un lado, realizan las tareas propias dentro del procesado de gráficos (*shading*, físicas, texturas, teselados, etc.). Por el otro, poseen unidades de *load/store* y de cálculo de funciones trascendentes e interpolación. Cada multiprocesador tiene una cantidad de memoria relativamente pequeña compartida entre los *cores* que lo forman. Además, existe una memoria global que está compartida por toda la GPU. Cuando un dato es transferido entre la CPU y la GPU, por defecto es almacenado o extraído de la memoria global.

Las función que permite el paso de memoria entre el *host* (CPU) y el *device* (GPU) o viceversa es `cudaMemcpy`. Para poder realizar la transferencia de memoria entre

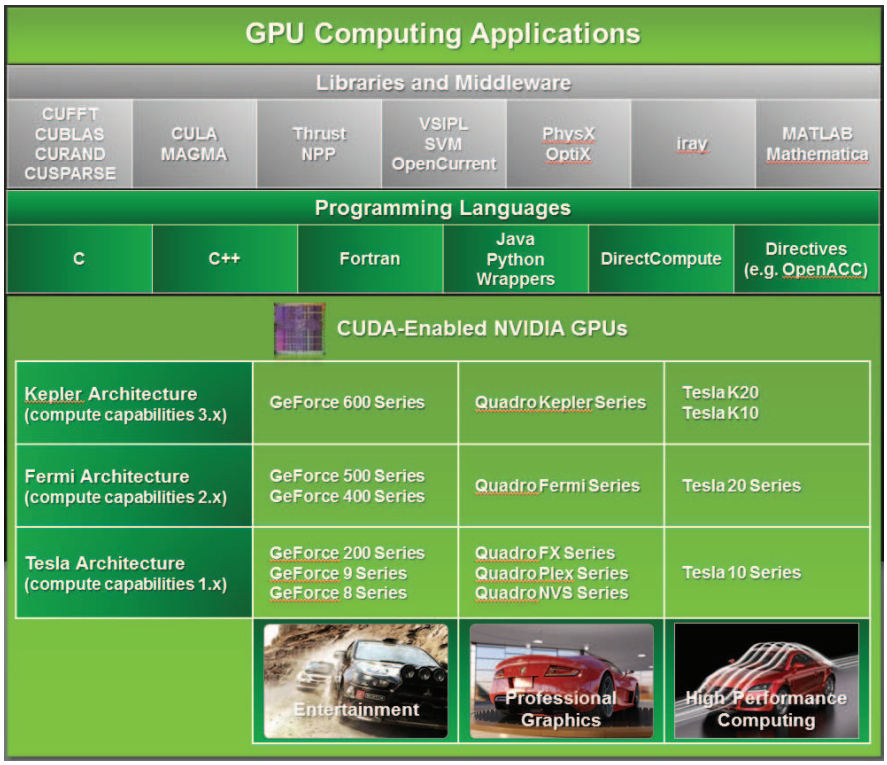

Figura 2.5: Arquitectura del entorno **CUDA®**

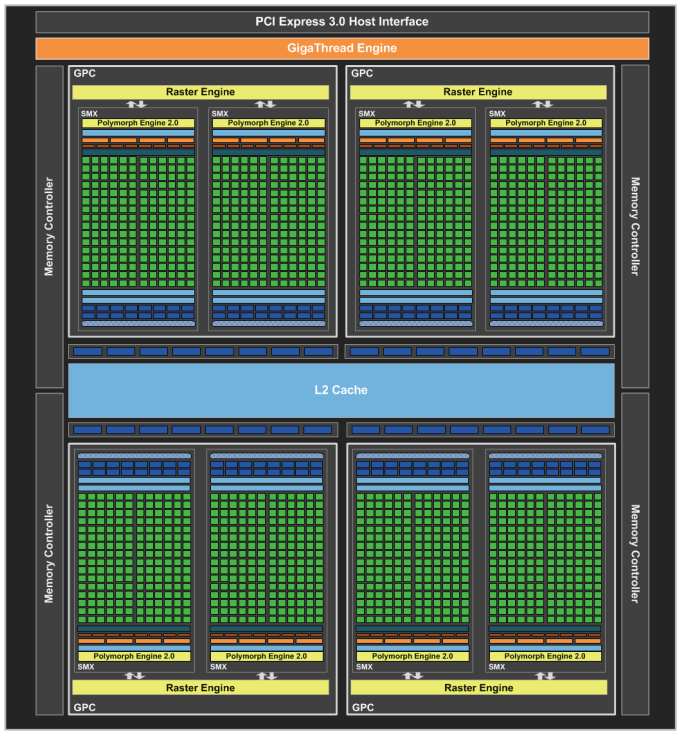

Figura 2.6: Arquitectura Nvidia Kepler

dos extremos es necesario haber reservado memoria tanto en origen como en destino mediante las funciones:

1. *Host.*

- `malloc`: asigna el número especificado de bytes.
- `calloc`: como `malloc`, además de inicializar a cero.
- `cudaMallocHost`: reserva la memoria (*pinned memory*) directamente sobre la RAM (*Random-Access Memory*) de forma que no requiere del *host* para realizar la transferencia por DMA (ver Figura 2.7).

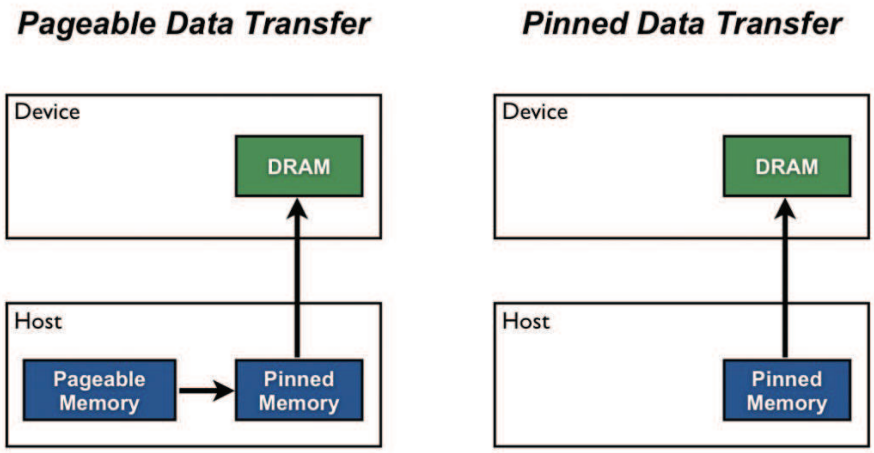

Figura 2.7: Transferencias de memoria entre el *host* y el *device*

2. *Device*.

- `cudaMalloc`: reserva memoria en la GPU.

Una vez se ha inicializado la memoria correctamente en la GPU, se invoca al *kernel* que no es más que una función en C que se ejecuta tantas veces como hilos se hayan reservado en la tarjeta gráfica, en lugar de una única vez como en el procesador. Para indicar que la función definida en el código es un *kernel*, se emplea el atributo `__global__`, mientras que para llamarlo la notación es la siguiente:

```
kernel<<<numBlocks, threadsPerBlock>>>
```

La jerarquía existente en la arquitectura de la GPU, tiene su análogo a nivel de **CUDA®** con los bloques y los hilos como se muestra en la Figura 2.8. Cada vez que se comienza un *kernel*, es necesario especificar el número de bloques y el número de hilos por bloque. En las arquitecturas actuales ambos números pueden tomar la forma de ternas de hasta 3 elementos.

Un *kernel* no puede, a su vez, llamar a funciones declaradas en el espacio de *host*. Para ello existe otro atributo, `__device__`. En el caso de que una función deba ser llamada por ambos extremos se debe emplear dos atributos, `__device__` y `__host__`. Los SMX ejecutan los hilos de forma SIMT (*Single Instruction, Multiple Threads*), donde todos los *cores* de un mismo grupo (*warp*, agrupación de 32 hilos) ejecutan la misma operación a la vez. En el caso de que existan hilos con saltos condicionales o que terminen antes que otros, algunos núcleos son desactivados, redundando de forma negativa en el rendimiento [22].

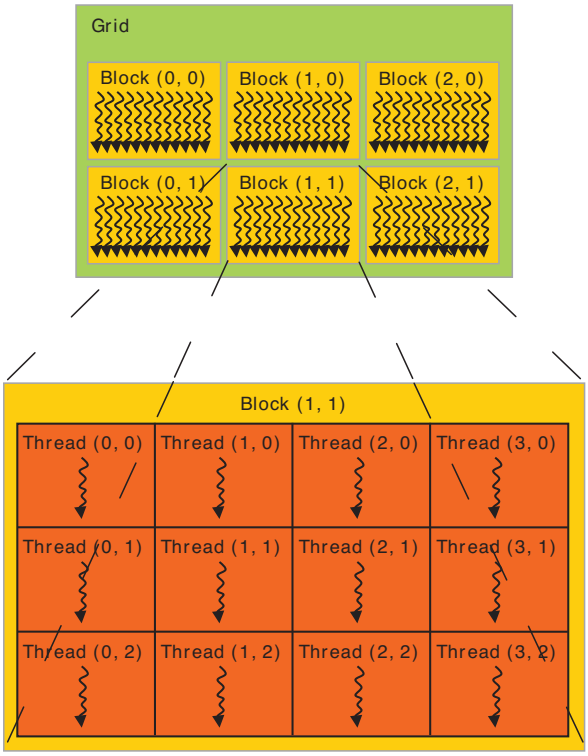

Figura 2.8: Jerarquía de bloques e hilos

Capítulo 3

Modelo matemático

Este capítulo describe el algoritmo seguido para implementar el simulador objeto del TFM. El modelo matemático parte de un preprocesado que permite aumentar la densidad de rayos útiles para el cálculo del funcional presentado en la Ecuación 1.1. A partir de ahí, el modelado del resto de componentes del escenario sigue los cauces habituales.

3.1. Justificación del uso de mililentes

En [2] la cámara empleada fue una **RED ONE®** [23] (ver Figura 3.1) con una resolución de 5120 (h) × 2700 (v) píxeles, donde el tamaño del sensor **Mysterium™** empleado es de 27,7 mm × 14,6 mm. Esto unido a que el *pitch* del *array* de microlentes era de tan solo 130 μm, permitía que bajo cada microlente hubiese aproximadamente 24 píxeles por dimensión. Si utilizáramos el mismo *array* con el fin de transmitir, nos encontraríamos con el problema de que los ledes son un orden de magnitud mayores a los fotodiodos. Por ejemplo, si tomamos como referencia una de las mejores pantallas OLED (*Organic Light-Emitting Diode*) existentes en el mercado, como la que monta el Samsung Galaxy S6, nos encontramos que, con una densidad de 577 PPI, el tamaño de píxel es de 44 μm. Empleando el mismo *array* únicamente contaríamos con 3 píxeles por dimensión lo cual es insuficiente a todas luces. Por este motivo, se decidió emplear mililentes con un *pitch* de 1,5 mm para poder alojar un número mayor de elementos.

3.2. Óptica geométrica

Como se avanzó en la Sección 1.2, el modelado físico del problema se corresponde con un problema de trazado de rayos. Esto significa que la propagación de la luz obedece a los fenómenos de reflexión y refracción. En la Figura 3.2 se muestra la interfaz entre dos materiales con índices de refracción η_1 y η_2.

La dirección del rayo incidente viene dada por el vector unitario, $\mathbf{i}$; de la misma forma, las direcciones de los rayos reflejado y refractado son $\mathbf{r}$ y $\mathbf{t}$, respectivamente.

Figura 3.1: **RED ONE®**

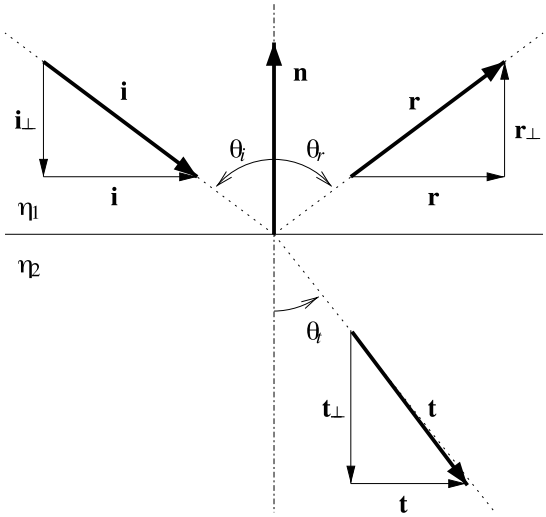

Figura 3.2: Ley de Snell

Igualmente, se define el vector normal, $\mathbf{n}$, perpendicular a la interfaz de los dos materiales. Por lo tanto, todos los vectores cumplen la condición de la Ecuación 3.1.

$$||\mathbf{i}|| = ||\mathbf{r}|| = ||\mathbf{t}|| = ||\mathbf{n}|| = 1 \tag{3.1}$$

Cada vector, a su vez, puede descomponerse en sus componentes normal y tangencial. Se emplea la notación $\mathbf{v}_\perp$ para denotar la componente normal del vector $\mathbf{v}$ y lo propio con $\mathbf{v}_\parallel$ para la componente tangencial. La componente normal de un vector se obtiene a partir de la proyección de este sobre el vector normal, tal y como se muestra en la Ecuación 3.2.

$$\mathbf{v}_\perp = (\mathbf{v} \cdot \mathbf{n})\,\mathbf{n} \tag{3.2}$$

Por lo que la componente tangencial puede obtenerse restando al vector original la componente normal, como se muestra en la Ecuación 3.3.

$$\mathbf{v}_\parallel = \mathbf{v} - \mathbf{v}_\perp \tag{3.3}$$

Por definición, el producto escalar entre ambas componentes es nulo, lo que quiere decir que ambas componentes son ortogonales entre sí; por consiguiente, se cumple la condición de la Ecuación 3.4.

$$||\mathbf{v}||^2 = ||\mathbf{v}_{\parallel}||^2 + ||\mathbf{v}_{\perp}||^2 \tag{3.4}$$

Los ángulos de incidencia, reflexión y refracción se denominan θ_i, θ_r y θ_t; y se definen siempre como el menor ángulo positivo entre la dirección del rayo y el vector normal. Mediante trigonometría se obtiene las relaciones de la Ecuación 3.5.

$$\begin{aligned} \cos\theta_v &= ||\mathbf{v}_{\perp}|| = \pm\mathbf{v}\cdot\mathbf{n} \\ \sin\theta_v &= ||\mathbf{v}_{\parallel}|| \end{aligned} \tag{3.5}$$

3.2.1. Reflexión

De los dos fenómenos, reflexión y refracción, la reflexión es el más sencillo y modela el choque mecánico de un rayo sobre una superficie reflectora. La ley de la reflexión nos dice que el ángulo de incidencia es igual al ángulo de reflexión: $\theta_r = \theta_i$. Si se desarrolla dicha igualdad como en la Ecuación 3.6.

$$\begin{aligned} ||\mathbf{r}_{\perp}|| &= \cos\theta_r = \cos\theta_i = ||\mathbf{i}_{\perp}|| \\ ||\mathbf{r}_{\parallel}|| &= \sin\theta_r = \sin\theta_i = ||\mathbf{i}_{\parallel}|| \end{aligned} \tag{3.6}$$

Por simple inspección de la Figura 3.2 se puede deducir la Ecuación 3.7.

$$\begin{aligned} \mathbf{r}_{\perp} &= -\mathbf{i}_{\perp} \\ \mathbf{r}_{\parallel} &= \mathbf{i}_{\parallel} \end{aligned} \tag{3.7}$$

Finalmente, la dirección del rayo reflejado dada por el vector $\mathbf{r}$ no será más que la Ecuación 3.8.

$$\begin{aligned} \mathbf{r} &= \mathbf{i}_{\parallel} - \mathbf{r}_{\perp} \\ &= [\mathbf{i} - (\mathbf{i}\cdot\mathbf{n})\mathbf{n}] - (\mathbf{i}\cdot\mathbf{n})\,\mathbf{n} \\ &= \mathbf{i} - 2(\mathbf{i}\cdot\mathbf{n})\mathbf{n} \end{aligned} \tag{3.8}$$

3.2.2. Refracción

La refracción se basa, casi en su totalidad, en la ley de Snell presentada en la Ecuación 3.9. Esta nos dice que el producto de los índices de refracción y los senos de los ángulos de ambos medios debe ser igual.

$$\eta_1 \sin\theta_i = \eta_2 \sin\theta_t \Rightarrow \sin\theta_t = \frac{\eta_1}{\eta_2}\sin\theta_i \tag{3.9}$$

De la Ecuación 3.9 se deduce inmediatamente que cuando el $\sin\theta_i > \frac{\eta_2}{\eta_1}$, el $\sin\theta_t$ debería ser mayor a la unidad; lo cual es imposible. En ese caso, se produce lo que se conoce como reflexión total interna. Al igual que con la reflexión, se descompone el rayo refractado en sus partes tangencial y normal. Aplicando las Ecuaciones 3.5 y 3.9, obtenemos la Ecuación 3.10.

$$||\mathbf{t}_\parallel|| = \frac{\eta_1}{\eta_2}||\mathbf{i}_\parallel|| \tag{3.10}$$

Las componentes tangenciales de ambos rayos son paralelas por lo que si al rayo incidente le sustraemos la parte normal (Ecuaciones 3.3), nos queda la Ecuación 3.11.

$$\mathbf{t}_\parallel = \frac{\eta_1}{\eta_2}\left[\mathbf{i} + \cos\theta_i \mathbf{n}\right] \tag{3.11}$$

Teniendo en cuenta que siempre se está trabajando con vectores unitarios y aplicando el teorema de Pitágoras, la componente normal se obtiene en la Ecuación 3.12.

$$\mathbf{t}_\perp = -\sqrt{1 - ||\mathbf{t}_\parallel||^2}\mathbf{n} \tag{3.12}$$

Uniendo las Ecuaciones 3.3 y 3.12, además de aprovechar las relaciones entre los ángulos y los rayos; se puede omitir el cálculo de funciones trigonométricas de forma explícita, según puede verse en la Ecuación 3.13.

$$\mathbf{t} = r\mathbf{i} + \left(rc - \sqrt{1 - r^2\left(1 - c^2\right)}\right)\mathbf{n} \tag{3.13}$$

donde $r = \frac{\eta_1}{\eta_2}$ y $c = -\mathbf{n}\cdot\mathbf{i}$.

3.2.3. Ángulo crítico

En la Subsección anterior se nombró la reflexión total interna como un fenómeno que provoca que no exista rayo refractado, por lo tanto, no hay transmisión de potencia al segundo medio. La condición para que esto no ocurra está implícita en el cálculo del rayo refractado. Si el radicando de la Ecuación 3.13 es negativo, significa que el rayo no se propaga. Se obvia el cálculo del ángulo crítico para no generar rayos que no se transmitan porque en la Sección 3.5 se combina con otra condición propia del problema.

3.2.4. Ecuaciones de Fresnel

De toda la luz que llega a una interfaz entre dos materiales no absorbentes: una parte se refleja de nuevo hacia el medio del que proviene, mientras que el resto se transmite hacia el segundo medio. De forma matemática esto se traduce en la Ecuación 3.14.

$$T + R = 1 \tag{3.14}$$

donde T y R significan transmitancia y reflectividad, respectivamente. La cantidad de luz reflejada o transmitida depende de los índices de refracción y del ángulo de incidencia. Las ecuaciones de Fresnel describen las amplitudes de las ondas reflejada y refractada en función de la amplitud de la onda incidente. En general, la reflectividad para luz polarizada se presenta en la Ecuación 3.15.

$$\begin{aligned} R_{\perp}(\theta_i) &= \left(\frac{\eta_1 \cos\theta_i - \eta_2 \cos\theta_t}{\eta_1 \cos\theta_i + \eta_2 \cos\theta_t} \right)^2 \\ R_{\parallel}(\theta_i) &= \left(\frac{\eta_2 \cos\theta_i - \eta_1 \cos\theta_t}{\eta_2 \cos\theta_i + \eta_1 \cos\theta_t} \right)^2 \end{aligned} \tag{3.15}$$

No obstante, al trabajar con luz no polarizada simplemente se promedia la reflectividad para ambas polarizaciones. Con todo, las expresiones finales para la reflectividad y la transmitancia pueden observarse en la Ecuación 3.16.

$$\begin{aligned} R(\theta_i) &= \begin{cases} \frac{R_{\perp}(\theta_i)+R_{\parallel}(\theta_i)}{2} & \text{si no hay reflexión total interna} \\ 1 & \text{si hay reflexión total interna} \end{cases} \\ T(\theta_i) &= 1 - R(\theta_i) \end{aligned} \tag{3.16}$$

3.3. Fuentes ópticas

Las principales fuentes empleadas en los sistemas de comunicaciones ópticas son el diodo láser y el led. Su estructura básica, en ambos casos, es la heterounión o lo que es lo mismo, la unión de dos semiconductores con energías de *gap* distintas. La región de emisión es una unión PN (*Positive-Negative*) de semiconductores III–V de *gap* directo que al ser polarizada en directa provoca que los portadores mayoritarios se difundan y recombinen, emitiendo energía en forma de luz (proceso radiativo) o disipándose en forma de calor (proceso no radiativo).

La principal diferencia entre los diodos láser y los ledes es la coherencia de la luz emitida. La radiación producida por un diodo láser se genera en una cavidad resonante, lo que le confiere una alta coherencia tanto espacial como temporal. Esto se traduce en una gran monocromaticidad y una alta directividad, siendo especialmente indicados para los enlaces punto a punto. Por contra, en un led no hay tal cavidad y la radiación

resultante tiene una anchura espectral considerable y además no coherente. La mayoría de las fuentes ópticas del mercado presentan diagramas de radiación lambertianos del estilo de la curva de la Figura 3.3 y responden a la Ecuación 3.17.

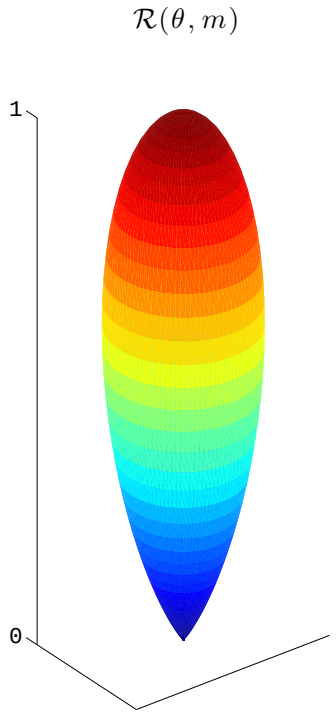

Figura 3.3: Diagrama de radiación lambertiano para $m = 20$

$$\mathcal{R}(\theta, m) = \frac{m+1}{2\pi} P_T \cos^m(\theta) \ \text{W/sr} \tag{3.17}$$

En este trabajo únicamente se ha considerado fuentes tipo led monocromáticas. Debido a que se pretende obtener el funcional descrito en la Ecuación 1.1, se necesita generar rayos en todo el ángulo sólido radiado por el emisor. Es por ello que se ha decidido mallar el espacio de forma regular y determinista para obtener información espacial relevante al problema. La dirección de salida de un rayo puede ser determinada por dos ángulos definidos en coordenadas esféricas: θ y ϕ. En un principio, se pensó barrer todo el ángulo sólido, sin embargo, este hecho incurre en una gran cantidad de rayos que sufren de reflexión total interna en la interfaz de salida de la lente; por lo tanto, en la Sección 3.5 se introduce el método optimizado de generación de rayos. En cualquier caso, un led queda definido por lo siguientes parámetros:

- Potencia total de emisión: P_T.
- Número de niveles en elevación: 2^T.
- Número de niveles en azimut: 2^P.
- Índice del patrón lambertiano: m.

3.4. Lentes

Una lente es cualquier objeto que sea capaz de desviar los rayos de luz mediante refracción. Además, se puede usar para enfocar la luz mientras que un prisma, a pesar de que también refracta la luz, no la enfoca. Los *arrays* de lentes vistos, tanto en la literatura como en el mercado, para aplicaciones relacionadas con imagen integral son mayoritariamente planoconvexos, donde una superficie de la lente es plana y la otra sobresale hacia afuera como la presentada en la Figura 3.4. La diferencia entre los distintos *arrays* viene dada por el perfil utilizado: cilíndrico, esférico, hexagonal, etc.

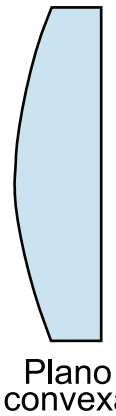

Figura 3.4: Lente planoconvexa

A la hora de modelarse el *array* de mililentes, se ha hecho uso de un modelo real como el MLA150-5C (perfil esférico) de Thorlabs [24] que es muy similar al empleado en el grupo CAFADIS. En particular, este modelo cuenta con una máscara que imposibilita el paso de la luz si no es a través de la mililente, aumentando así el contraste (ver Figura 3.5). Este hecho será clave en la Sección 3.5. Los parámetros necesarios en la definición de una lente se muestran en la Figura 3.6.

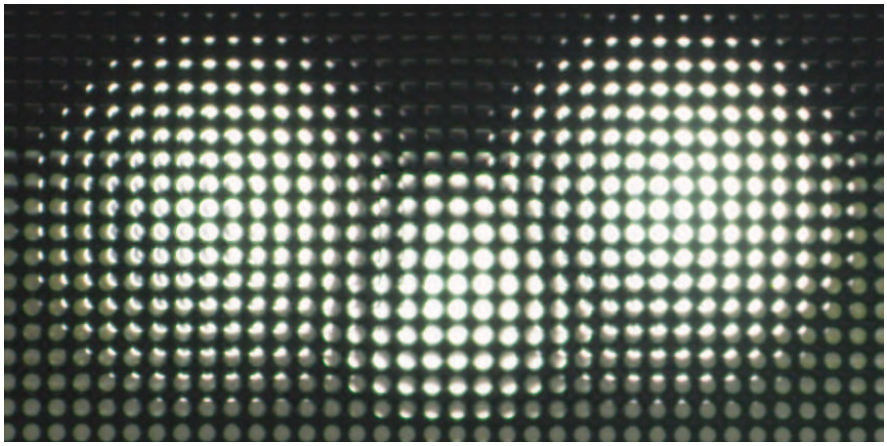

Figura 3.5: Imagen captada con un *array* de microlentes encima del sensor

- Material (índice de refracción): n.
- Perfil convexo: esférico.

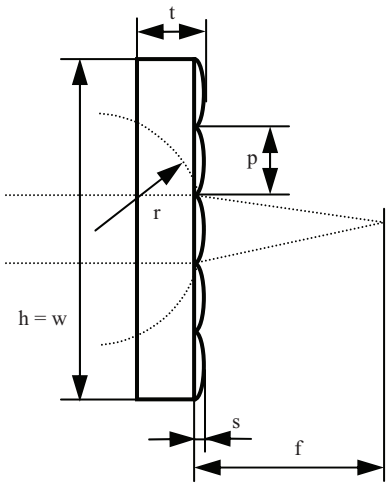

Figura 3.6: MLA150-5C

- Diámetro de la lente: d.
- Altura y ancho: h y w.
- Espesor: t.
- Espaciado: p.
- Casquete esférico: s.
- Radio de la esfera: r.
- Distancia focal: f.

3.5. Preprocesado

El modelo matemático presentado a lo largo del Capítulo 3 es perfectamente válido para ser llevado a la práctica, sin embargo, tras una primera implementación, se observó que el número de rayos que atravesaban las mililentes era escaso en comparación al número total de rayos generados. Esto se debe a dos razones fundamentales: por un lado, la geometría del problema, y por el otro, la máscara de las mililentes empleadas. A pesar de que la fuente óptica radie sobre un ángulo mitad relativamente grande, el ángulo sólido efectivo que alcanza la mililente es bastante menor. Si se obtuviese el ángulo sólido efectivo antes de generar los rayos, se podría aumentar el porcentaje de rayos útiles para el objetivo del trabajo. Es por ello que se decidió añadir al modelo matemático un preprocesado que tuviese en cuenta este hecho, para simplificar, se dividió el proceso en dos pasos: azimut y elevación.

3.5.1. Preprocesado en azimut

Como se comentó en la Sección 3.3, la dirección de salida de un rayo está descompuesta en los ángulos definidos en esféricas: ϕ y θ. En ambos casos, la solución

pasa por abatir las ecuaciones sobre dos de las tres dimensiones. Para el caso que nos ocupa, ϕ, suponiendo que las fuentes se encuentran en el plano XY y que las mililentes se encuentran sobre estas desplazadas una determinada distancia sobre el eje Z (ver Figura 3.7), basta proyectar sobre el plano XY.

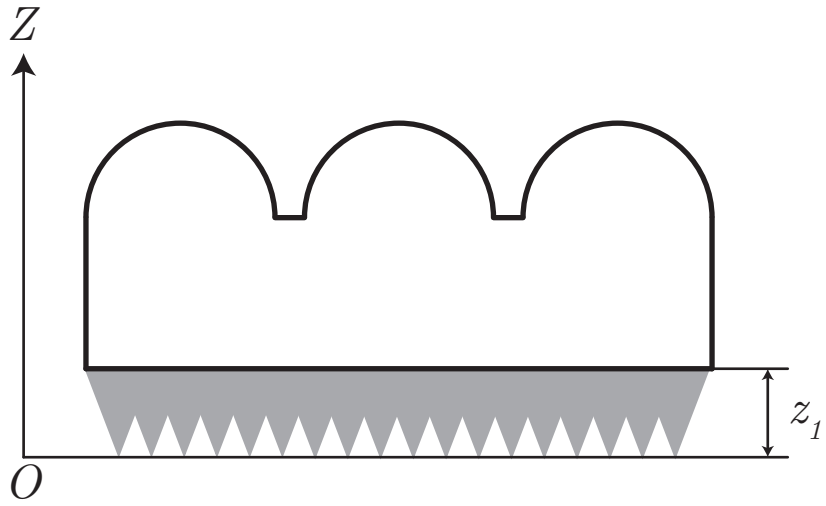

Figura 3.7: Situación de partida

Al proyectar sobre dicho plano, el problema se reduce a encontrar los ángulos tangentes a una circunferencia como puede verse en la Figura 3.8. En primer lugar, el vector resultante de la resta del centro de una mililente y una fuente forma la hipotenusa de dos triángulos rectángulos iguales. Estos triángulos están formados por la unión de la recta tangente de la fuente a la circunferencia de la mililente y el radio de la circunferencia que pasa por el punto de tangencia. Una vez calculados el ángulo central y el semiángulo correspondiente, basta sumarlos o restarlos para obtener los valores máximo y mínimo, respectivamente. De forma matemática, en las Ecuaciones 3.18.

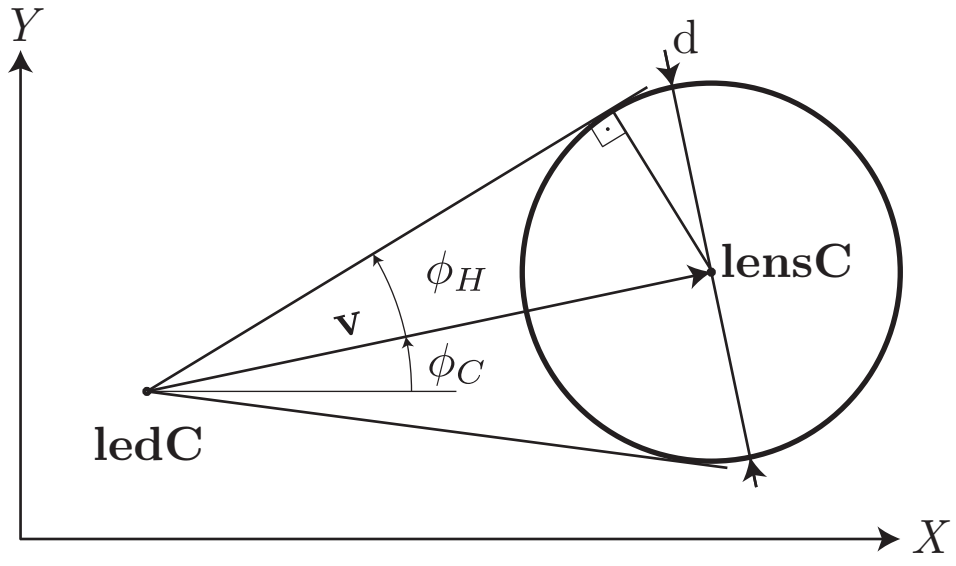

Figura 3.8: Preprocesado en azimut

$$\begin{aligned} \mathbf{v} &= \mathbf{lensC} - \mathbf{ledC} \\ D &= ||\mathbf{v}|| \\ \phi_C &= \arg(\mathbf{v}) \\ \phi_H &= \arcsin\left(\frac{d}{2D}\right) \\ \phi_{\text{máx,mín}} &= \phi_C \pm \phi_H \end{aligned} \tag{3.18}$$

El único caso que no cumple lo anteriormente dicho es cuando la fuente se encuentra a una distancia de la mililente inferior al radio de la misma, o lo que es lo mismo está dentro de la circunferencia, en ese caso, el rango de ángulos útiles cubre todo el dominio $[0, 2\pi)$.

3.5.2. Preprocesado en elevación

Una vez determinados los límites en azimut, se puede barrer dicha dimensión sin mayor problema. Para cada ángulo dentro del rango, sabemos por la ley de Snell 3.9 que el rayo incidente, el rayo refractado y la normal a la interfaz están contenidos en un mismo plano; por lo tanto, el rayo refractado tendrá el mismo azimut que el rayo incidente. Sin embargo, para obtener la elevación máxima y mínima posible para cada caso se ha dividido, a su vez, el proceso en dos pasos.

En el primero, se proyecta de nuevo sobre el plano XY, reduciendo el problema a encontrar el corte entre la recta con dirección ϕ_i y la circunferencia, tal y como se muestra en la Figura 3.9. Con esto se obtiene la distancia máxima y mínima, sobre el plano XY, que puede recorrer un rayo proveniente de una fuente al impactar con una determinada mililente. En las Ecuaciones 3.19 se presentan las expresiones matemáticas vinculadas a esta parte del problema.

$$\begin{aligned} ||\mathbf{x} - \mathbf{lensC}||^2 &= r^2 \\ \mathbf{x} &= \mathbf{ledC} + s\mathbf{i} \\ \mathbf{v} &= \mathbf{lensC} - \mathbf{ledC} \\ ||s\mathbf{i} - \mathbf{v}||^2 &= r^2 \\ \xi &= r^2 - \mathbf{v}^2 \\ \eta &= \mathbf{i} \cdot \mathbf{v} \\ s_{\text{máx,mín}} &= \eta \pm \sqrt{\eta^2 + \xi} \end{aligned} \tag{3.19}$$

En el segundo paso, con $s_{\text{máx}}$ y $s_{\text{mín}}$ ya calculados, se tiene la situación planteada en la Figura 3.10 donde por trigonometría se puede plantear la Ecuación 3.20.

$$s_{\text{máx,mín}} = z_1 \tan\theta_1 + z_2 \tan\theta_2 \tag{3.20}$$

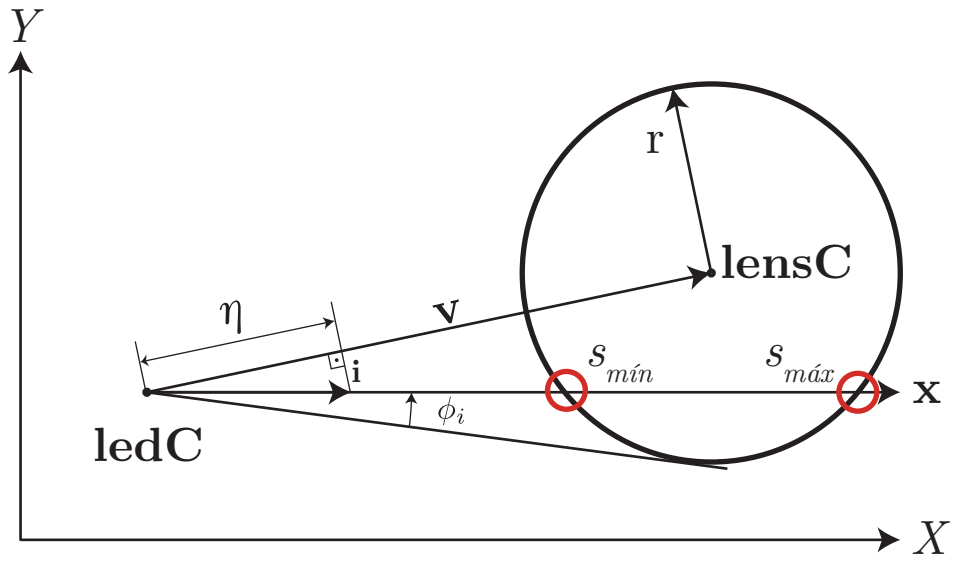

Figura 3.9: Preprocesado en elevación

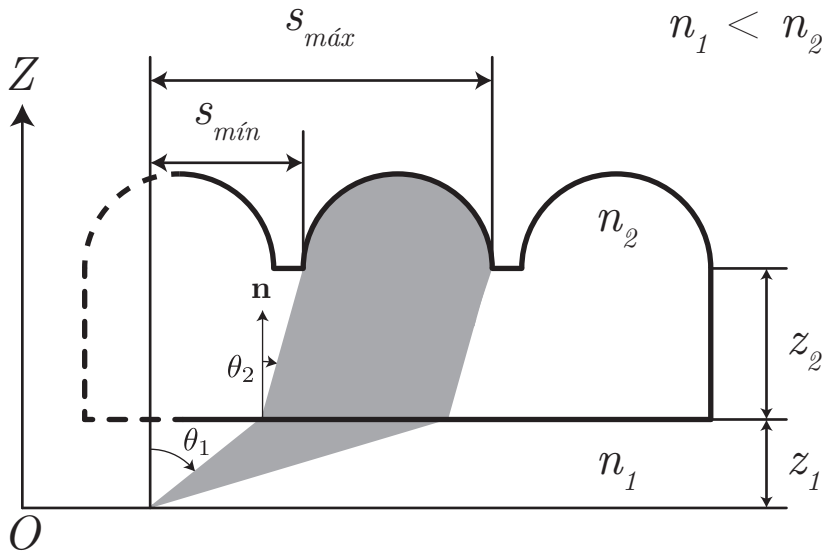

Figura 3.10: Cálculo de los ángulos de elevación

Transformando la Ecuación 3.20 para dejar todo expresado en términos de $\tan\theta_1$, que es el ángulo que necesitamos para generar los rayos, se obtiene la Ecuación 3.21 que es irresoluble por métodos analíticos. Llegados a este punto, cualquier método de cálculo numérico se puede emplear para su resolución. Por último, basta despejar el ángulo de la función trigonométrica, $\theta_1 = \arctan(\tan\theta_1)$.

$$s_{\text{máx,mín}} = z_1 \tan\theta_1 + z_2 \frac{n_1}{n_2} \frac{\tan\theta_1}{\sqrt{1+\tan^2\theta_1\left(1-\left(\frac{n_1}{n_2}\right)^2\right)}} \tag{3.21}$$

En nuestro caso, se ha decidido emplear el método de Newton-Raphson puesto que la Ecuación 3.21 presenta un intervalo de monotonía creciente muy similar a una

recta con variable independiente $\tan\theta_1$. En efecto, se muestra en la Figura 3.11 el comportamiento de dicha función.

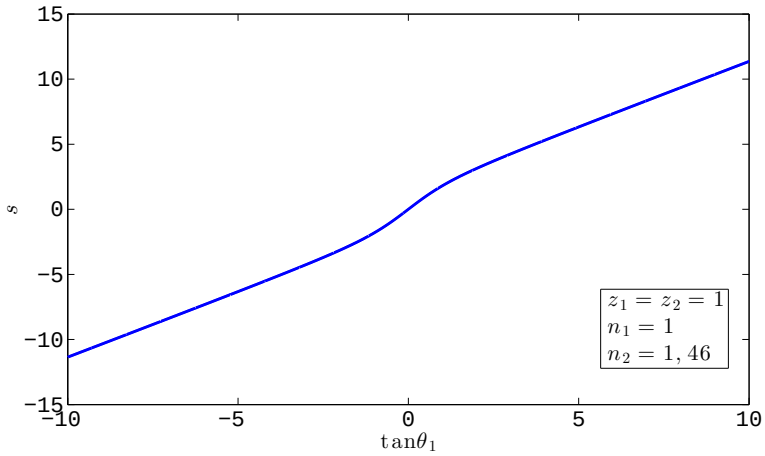

Figura 3.11: Función a resolver de forma numérica

Mediante este procesado se obtiene la 4-tupla: $\phi_{\text{mín}}$, $\phi_{\text{máx}}$, $\theta_{\text{mín}}$ y $\theta_{\text{máx}}$. Esta determina, para cada par fuente-mililente, el ángulo sólido que impacta sobre la base de la mililente, esto es, sobre la base de la protuberancia. De esta forma, se garantiza que todos los rayos generados alcanzan las mililentes y se ignoran todos los rayos que impactarían sobre la superficie absorbente existente entre las mismas. Esto no resta un ápice de validez al método de generación de rayos puesto que la energía de los rayos está ponderada por su dirección y con este proceso simplemente se está haciendo un muestreo selectivo. El resto de pérdidas son calculadas mediante las ecuaciones de Fresnel.

Capítulo 4

Implementación del sistema

En este capítulo se aborda tanto el diseño del código de forma genérica, así como las particularidades de cada una de las versiones, a saber: secuencial, memoria compartida y GPU. En primer lugar, se presenta la topología del problema y las estructuras de datos necesarias. Además, se describen las funciones desarrolladas y las decisiones tomadas para adaptar el modelo matemático.

4.1. Topología del problema

El escenario de partida consta de un *display*, de un *array* y un plano imagen. El objetivo, desde el punto algorítmico, es obtener un vector de rayos generados en la fuente y otro de rayos refractados sobre la superficie de las mililentes. De esta forma, se podrá realizar un ajuste de funciones que permita obtener el funcional descrito en la Ecuación 1.1. Igualmente, como se comentó en la Sección 1.2, el criterio de parada en un hipotético proceso de síntesis es un patrón de radiación a una distancia dada; de ahí, el tercer vector de impactos sobre el plano imagen.

En cualquier caso, todos los rayos siguen el mismo proceso: generación, propagación hasta las mililentes, refracción medio-lente, propagación intralente, refracción lente-medio y propagación hasta el plano. En el primer paso, se genera el vector de rayos generados; en la segunda refracción (lente-medio), se obtiene el vector de refractados; y por último, el vector de impactos una vez se propagan hasta una distancia conocida. De forma gráfica, en la Figura 4.1 se observa, de forma simplificada, los distintos pasos que sigue un rayo y cómo se generan los resultados del algoritmo.

4.2. Estructuras de datos

Puesto que el objetivo es obtener rayos en diferentes puntos del escenario, la primera estructura de datos necesaria es un registro que contenga la información relevante de un rayo. Dicha estructura, denominada `Ray`, cuenta con los campos de la Tabla 4.1.

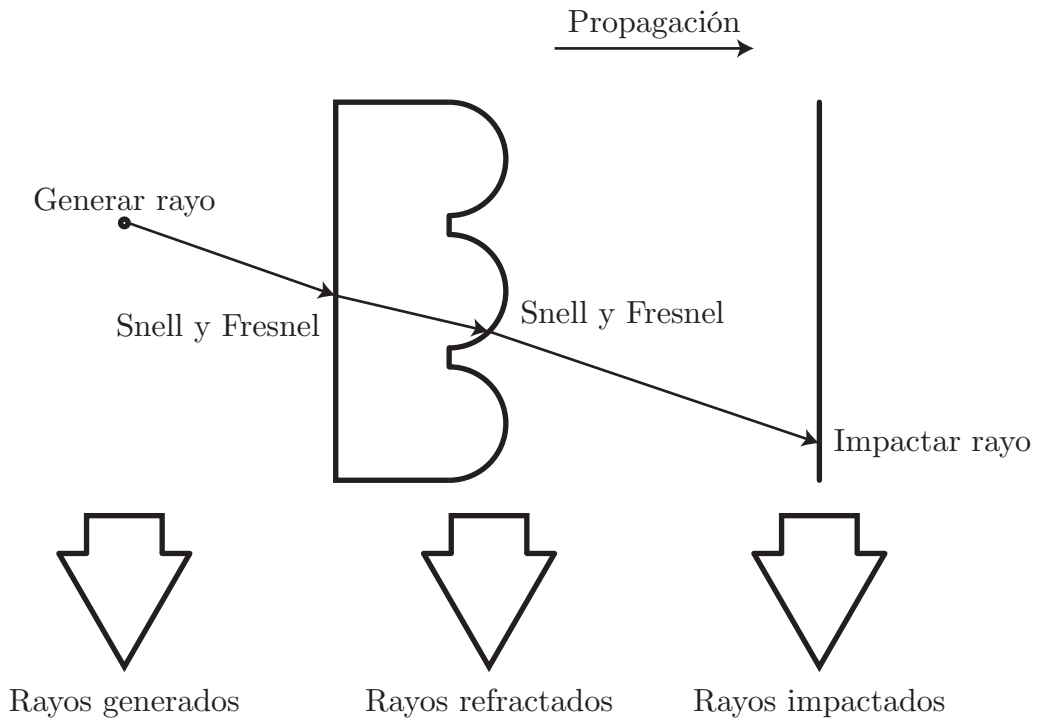

Figura 4.1: Diagrama del algorimo

`Ray`		
Campo	**Significado**	**Tipo de dato**
Posición	Coordenadas cartesianas de la posición del rayo.	`float/double`
Vector director	Vector unitario que define la dirección de propagación del rayo.	`float/double`
Distancia	Distancia recorrida desde la fuente.	`float/double`
Potencia	Potencia por unidad de ángulo sólido.	`float/double`
Etiqueta	Identifica la procedencia de cada rayo mediante su fuente de emisión.	`unsigned int`

Tabla 4.1: Estructura de datos para un rayo

Mediante esta abstracción en el código, la información de los ledes se traslada a los rayos durante la inicialización de los mismos. A partir de ahí, la única alusión a los ledes estará en la etiqueta que forma parte de los campos del registro `Ray`. En el caso de las mililentes, a pesar de que se utiliza una estructura similar, registros, se empleó una división jerárquica con vistas a flexibilizar la ampliación del simulador a otros tipos de lentes, distintas a las plano-convexas o modificando el perfil. El registro `Lens` (ver Tabla 4.2) consta de dos campos que son otros dos registros del tipo `Interface`: `Left` y `Right`. A su vez, la estructura `Interface` posee los campos de la Tabla 4.3.

`Lens`		
Campo	**Significado**	**Tipo de dato**
Left	Primera interfaz de la lente (medio-lente).	`struct Interface`
Right	Segunda interfaz de la lente (lente-medio).	`struct Interface`

Tabla 4.2: Estructura de datos para una lente

En este TFM, al emplear un modelo concreto de mililente, únicamente se han implementado los tipos plana y esférica. En estos casos, el parámetro indica el radio de la lente que puede ser infinito (interfaz plana), positivo (interfaz convexa) o negativo

`Interface`		
Campo	**Significado**	**Tipo de dato**
Posición	Coordenadas cartesianas del centro óptico.	`float/double`
Tipo	Especifica el perfil de la lente.	`unsigned int`
Parámetro	En función del tipo de interfaz tiene un significado diferente.	`float/double`
Índice de refracción	Material de fabricación.	`float/double`

Tabla 4.3: Estructura de datos para una interfaz

(interfaz cóncava). Si se añadiesen perfiles *ad-hoc*, el parámetro podría tomar el significado que se desease. Para finalizar, un escenario de simulación se define a través de una serie de parámetros, los cuales son:

- Distancia a las mililentes y al plano de impacto.
- Índices de refracción de los medios.
- Número de ledes y mililentes.
- Separación, diámetro y espesor de las mililentes.
- Tamaño de la protuberación (cúpula) y radio de curvatura de las mililentes.
- Índice del emisor lambertiano y potencia total de un emisor.
- Número de niveles en azimut y elevación por pareja de led-mililente.

4.3. Explotación de la simetría

Antes de abordar la descripción del algoritmo, se hace necesario nombrar la estrategia seguida para simular. A pesar de que no se ha nombrado en ningún momento cuántos ledes forman un *display* o cuántas mililentes hay por pulgada, el número de elementos se dispara a poco que se introduzcan valores comerciales. Asimismo, si se supone que los ledes y las mililentes están alineados, es decir, una fila de ledes es paralela a otra de mililentes; se puede comprobar que el problema es básicamente repetitivo.

Por este motivo, se replanteó la idea inicial de simular toda la pantalla y, en su lugar, tomar sólo una parte representativa de la misma de forma que, posteriormente, por superposición pueda recomponerse el patrón de radiación completo. Para ello se planteó agrupar las mililentes en clústeres según su conectividad (vecindad), empleando 8-conectividad. De esta forma los clústeres pueden contener 1, 9, ..., $(2{\cdot}n{+}1)^2$ mililentes, siendo $n \in \mathbb{N}$ un parámetro de simulación. El tamaño óptimo del clúster para que la simulación sea precisa se podría determinar mediante un criterio geométrico, en función de la potencia emitida en los ángulos sólidos subtendidos bajo las mililentes, no obstante, en el presente TFM, se hace un pequeño estudio numérico de la precisión en la Sección 5.2. De igual forma, los ledes pueden agruparse sobre la mililente central del clúster, cubriendo todos los casos posibles. En este caso, el número de ledes sigue potencias pares de dos según 2^{2m}, con $m \in \mathbb{N}$. En la Figura 4.2 se presenta la idea de agrupar las mililentes y los ledes para agilizar el cálculo.

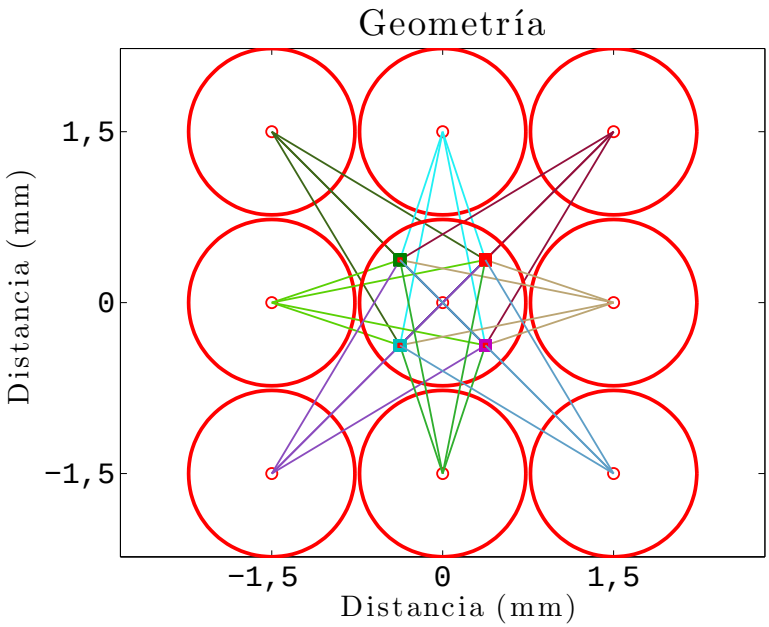

Figura 4.2: Clúster de 9 mililentes y 4 ledes por mililente

Siguiendo este razonamiento, se podría seguir explotando la simetría del problema y calcular un único tramo $\left[0, \frac{\pi}{4}\right]$ sobre la dimensión ϕ y aplicar simetría de revolución para cubrir todos los ángulos. No obstante, se descartó llevar hasta el límite la aplicación de simetrías puesto que para síntesis no aporta ninguna ventaja. En cambio, manejar un clúster puede resultar conveniente llegado el caso.

4.4. Descripción del algoritmo

En el Algoritmo 1 se introduce el pseudo-código del simulador completo. De forma general, el código puede dividirse en las siguientes subsecciones: definición del escenario, cálculo de límites y simulación.

4.4.1. Definición del escenario

En primer lugar, se leen los parámetros de la simulación a través de la función `ReadConfiguration`. A partir de los parámetros de entrada, se inicializan las estructuras `Lens` e `Interface` mediante las funciones `InitGeometry` e `InitScenario`. `InitGeometry` calcula los centros de las mililentes y los ledes en función del clúster y la densidad de fuentes deseada, mientras que `InitScenario` solicita la memoria para las estructuras que alojan la información de las mililentes y las inicializa.

Algorithm 1 Simulador

```
1: procedure LENSLET
2:     READCONFIGURATION (file in, params)
3:     leds ← 2^(2m)
4:     lenses ← 2 · n + 1
5:     INITGEOMETRY (ledsCenters, lensesCenters)
6:     INITSCENARIO (array)
7:     RAYSPROPAGATION (rays)
8:     WRITERESULTS (file out, rays)
9: end procedure

10: procedure RAYSPROPAGATION(rays)
11:     GETPHILIMITS (φ_mín, φ_máx)
12:     P ← 2^p
13:     T ← 2^t
14:     for i ← 1, leds do
15:         for j ← 1, lenses do
16:             dφ ← (φ_máx(i,j) − φ_mín(i,j)) / (P − 1)
17:             for k ← 1, P do
18:                 φ ← φ_mín(i, j) + k · dφ
19:                 GETSECONDORDERSOLUTIONS (s_mín, s_máx)
20:                 GETTHETAFROMDISTANCE (θ_mín, θ_máx)
21:                 dθ ← (θ_máx − θ_mín) / (T − 1)
22:                 for l ← 1, T do
23:                     θ ← θ_mín + l · dθ
24:                     GENERATERAY (rays(i, j, k, l), m, φ, θ, P_T)
25:                     CALCULATEOUTPUT (rays(i, j, k, l), array(j))
26:                     CALCULATEIMPACT (rays(i, j, k, l), distance)
27:                 end for
28:             end for
29:         end for
30:     end for
31: end procedure

32: procedure CALCULATEOUTPUT(ray, lens)
33:     CALCULATEPLANE (ray, left)
34:     SNELLFRESNEL (ray, left)
35:     CALCULATESPHERE (ray, right)
36:     SNELLFRESNEL (ray, right)
37: end procedure
```

4.4.2. Cálculo de límites

Tras realizar las acciones anteriores, donde únicamente se han definido las condiciones de contorno del escenario de simulación, se procede al preprocesado que se introdujo en la Sección 3.5. Tanto el cálculo de límites como la simulación están dentro de una rutina mayor denominada `RaysPropagation`. Primero, para cada par led-mililente se calculan los ángulos azimutales límites mediante la rutina `GetPhiLimits`. Según se barre los ángulos ϕ_i, se calculan los ángulos de elevación máximo y mínimo en dos pasos con las rutinas `GetSecondOrderSolutions` y `GetThetaFromDistance`. Con esto se determinan los ángulos sólidos útiles desde el punto de vista de la simulación. En la Figura 4.3 se presenta un ejemplo para un clúster de 9 mililentes y un único led.

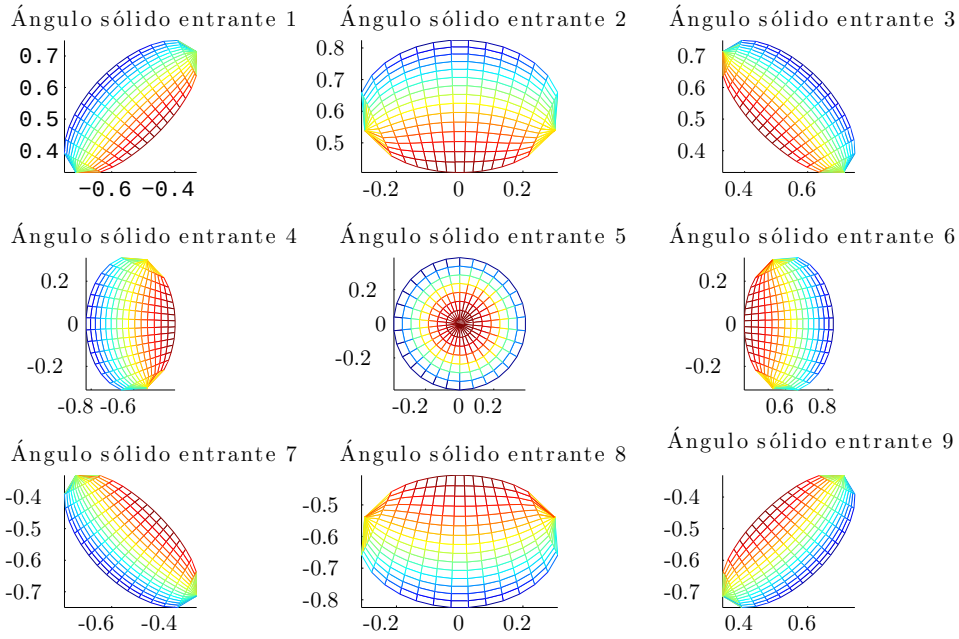

Figura 4.3: Malla de ángulos sólidos a simular sobre el plano XY

4.4.3. Simulación

Una vez la dirección de cada rayo queda prefijada con los pasos anteriores, se pasa a generar cada uno siguiendo el patrón de radiación lambertiano (modificable a través del parámetro m). A pesar de que la potencia de cada led también es un parámetro de la simulación, el preprocesado introducido fuerza a que no se genere todo el diagrama de radiación completo y con ello, si se realizase la integral sobre los rayos generados, no se obtendría la potencia introducida. La diferencia entre el valor introducido y el de la integral se corresponde con las pérdidas por absorción en la máscara de las mililentes, mientras que las pérdidas por reflexión total interna se calculan mediante las ecuaciones de Fresnel. Si se cambiase el tipo de mililentes a uno sin máscara, bastaría con desechar el preprocesado y generar todos los rayos. El proceso siguiente se realiza dentro de un

mismo bucle, excepto los rayos directos de cada fuente que se calculan en un bucle externo. La simulación comprende los siguientes pasos:

- Generación de los rayos en las direcciones calculadas. Rutina `GenerateRay`.
- Cálculo del camino óptico hasta la salida de las mililentes a través de la rutina `CalculateOutput`. Esta, a su vez, se subdivide en:
  - Primero, impacto con la interfaz plana de las mililentes. Rutina `CalculatePlane`.
  - Seguidamente, refracción del medio original a las mililentes. Rutinas `SnellFresnel` o `SnellSchlick`.
  - Posteriormente, cálculo del impacto con la cúpula de las mililentes mediante la rutina `CalculateSphere`.
  - Por último, segunda refracción con las mismas rutinas `SnellFresnel` o `SnellSchlick`.
- Cálculo del impacto con el plano imagen. Rutina `CalculateImpact`.

Los resultados obtenidos se almacenan en un fichero externo para ser procesados en software externo al TFM con la rutina `WriteResults`. Un ejemplo de los rayos almacenados puede verse en la Figura 4.4 donde, para un clúster de 9 mililentes y 4 ledes, pueden verse los impactos sobre un plano situado a 3 metros del origen.

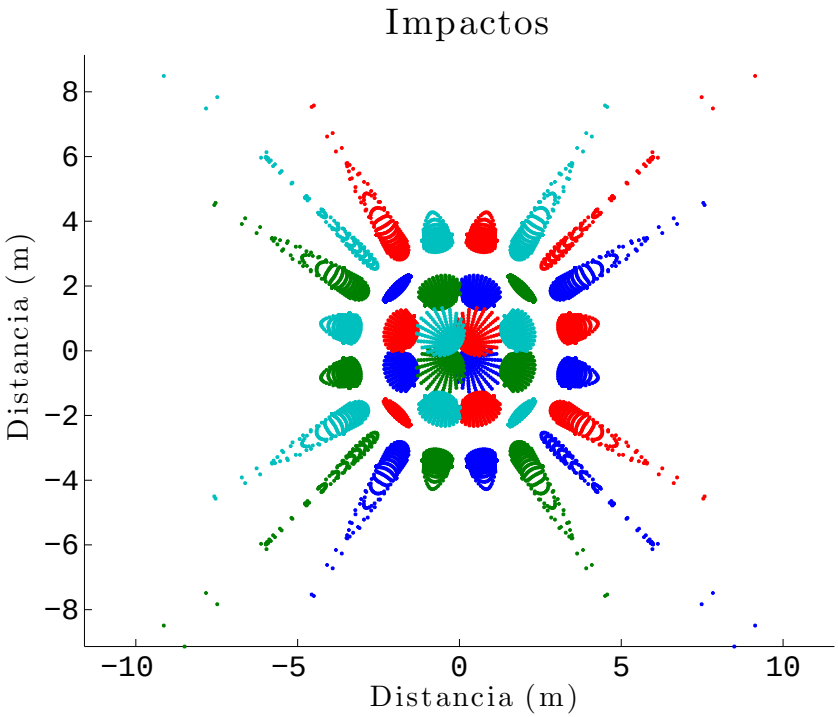

Figura 4.4: Impactos sobre el plano imagen

Rutinas de cálculo de impactos

En el escenario planteado para simular pueden ocurrir dos tipos de impactos: con un plano o con una esfera. El primero se da al impactar con la lente plano-convexa por su interfaz plana, así como con el plano imagen (rutina `CalculatePlane`). La segunda al salir de la mililente por la cúpula que no está cubierta por la máscara (rutina `CalculateSphere`). En ambos casos, se calcula el punto de impacto del rayo en coordenadas cartesianas y se actualiza tanto el punto de aplicación del rayo como la distancia recorrida. Estas distancias se calculan de la siguiente forma según las Ecuaciones 4.1 y 4.2.

$$d_{\text{plane}} = \frac{z_{\text{plane}} - z_{\text{ray}}}{\mathbf{v}_z} \tag{4.1}$$

$$d_{\text{sphere}} = \eta \pm \sqrt{\eta^2 + \xi} \tag{4.2}$$

En la Ecuación 4.1 se puede prescindir del valor absoluto puesto que la propagación de los rayos siempre se realiza sobre el eje Z positivo. No obstante, la Ecuación 4.2 es exactamente la misma que la Ecuación 3.18 puesto que el problema del corte de una recta con un círculo es equivalente al corte de una recta con una esfera, añadiendo una dimensión extra. En este caso, no interesa obtener las dos soluciones que nos devuelve el radicando sino la que sea mayor que cero. El significado físico de esto es que la distancia positiva es el corte con la cúpula, mientras que la distancia negativa, aunque es también un corte con la esfera, no existe en la realidad. Esto es así porque el rayo está *dentro* de la esfera. En caso contrario, ambas soluciones serían positivas.

Rutinas de cálculo de refracción

Con vistas a reutilizar cálculos se decidió unir el cálculo del rayo refractado con las ecuaciones de Fresnel, de este modo se evita calcular las funciones trigonométricas de forma reiterada (rutina `SnellFresnel`). Como la refracción depende de la normal en el punto de impacto, se debe diferenciar cuándo se impacta contra un plano o contra una esfera. El cálculo de normales se realiza como en las Ecuaciones 4.3 y 4.4.

$$\mathbf{n}_{\text{plane}} = \mathbf{e}_z \tag{4.3}$$

$$\mathbf{n}_{\text{sphere}} = \frac{\mathbf{x}_{\text{ray}} - \mathbf{lensC}}{||\mathbf{x}_{\text{ray}} - \mathbf{lensC}||} \tag{4.4}$$

A partir de aquí, se sigue con las ecuaciones presentadas en el Capítulo 3 para el rayo refractado (Ecuación 3.13) y las pérdidas de Fresnel (Ecuación 3.16). No obstante, en la literatura se encontró un método optimizado de cálculo de las ecuaciones de Fresnel: la aproximación de Schlick [25]. En las Ecuaciones 4.5 y 4.6 se resume sus expresiones.

$$R_{Schlick}(\theta_i) = \begin{cases} R_0 + (1 - R_0)(1 - \cos\theta_i)^5 & \eta_1 \leq \eta_2 \\ R_0 + (1 - R_0)(1 - \cos\theta_t)^5 & \eta_1 > \eta_2 \quad \text{si no hay reflexión total interna} \\ 1 & \eta_1 > \eta_2 \quad \text{si hay reflexión total interna} \end{cases} \tag{4.5}$$

con R_0:

$$R_0 = \left(\frac{\eta_1 - \eta_2}{\eta_1 + \eta_2}\right)^2 \tag{4.6}$$

En [26] se asegura que el cálculo de las pérdidas es hasta un 30 % más rápido en comparación a la ecuación de Fresnel para luz no polarizada si se evita el uso de funciones de exponenciación del tipo `pow`, en caso contrario, se vuelve el doble de lenta. Esto ocurre cuando se compara el cálculo aislado de ecuaciones de Fresnel, sin embargo, en nuestro caso se han integrado en una función que engloba tanto el cálculo de la refracción como el de las pérdidas, por lo que esta aceleración ya no resulta tan evidente.

4.5. Código secuencial

La situación de partida del TFM fue un prototipo secuencial desarrollado en **MATLAB®**. A partir de este prototipo, se reescribió en C el código secuencial. De partida, sabiendo que habría que portarlo en última instancia a **CUDA®**, se intentó, en la medida de lo posible, eliminar las posibles bifurcaciones en el código (estructuras `if-then-else` o `switch`) para minimizar la *warp divergence*. Esta sucede cuando no todos los hilos de un *warp* ejecutan la misma instrucción, normalmente cuando hay ramas condicionales. En este caso, cada rama del código se ejecuta de forma secuencial y los hilos que no cumplen la condición se desactivan. Además, se parametrizó el código de forma que pudiese cambiarse la precisión entre simple y doble mediante directivas `#define`. De esta forma, la estructura `Ray` ocupa en memoria 36 bytes cuando se trata de `float` y 72 bytes en el caso de `double`.

Además, se decidió cómo irían los rayos alojados en memoria para facilitar los accesos a la misma, esto es, que se hiciesen de forma secuencial y, en la medida de lo posible, alineados. En un primer momento, se crearon 3 *arrays* de rayos unidimensionales para los distintos resultados necesarios, aunque tras algunos cambios en el orden de los bucles buscando el mejor rendimiento posible se optó por crear un único *array* que englobase a todos. A esto hay que añadir que, dentro del preprocesado, por cada ángulo en azimut se generan varios ángulos en elevación por lo que esa jerarquía era recomendable mantenerla en memoria. Con todo, en la Figura 4.5 se muestra el formato del *array* de rayos.

Así, de mayor a menor orden jerárquico: primero, se subdivide en fuentes (ledes); seguidamente, en lentes; posteriormente, en azimut; después, en elevación y; por último,

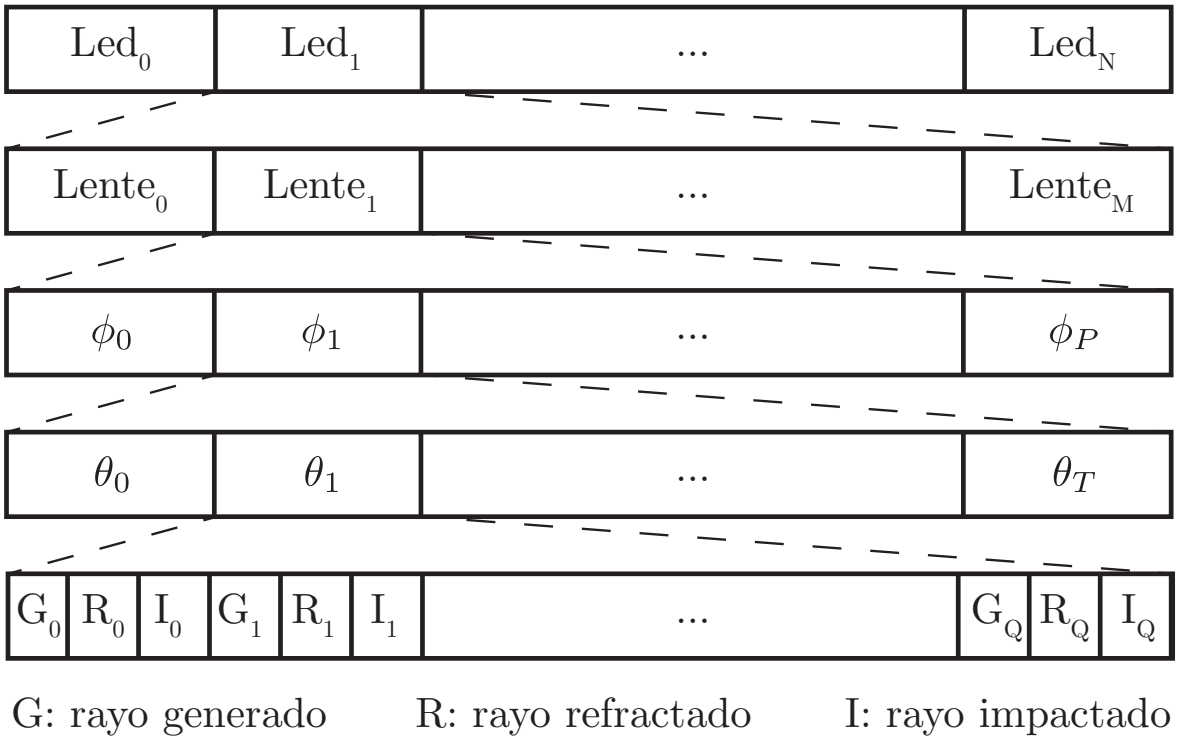

Figura 4.5: Formato del vector de rayos

el rayo en las tres posiciones en las que se desea obtener. De esta forma, según se recorre el *array* mediante un índice global se va poblando el mismo.

4.6. Código memoria compartida

Una vez comprobado que los resultados obtenidos por la versión secuencial en C se correspondían con aquellos obtenidos en **MATLAB®**, se hizo uso de un *profiler* incluido en el IDE (*Integrated Development Environment*) Xcode para localizar los *hot-spots* del código. El resultado se presenta en la Tabla 4.4.

Tiempo (%)	**Tiempo**	***Self***	**Nombre**
100 %	56477,1 ms	0 ms	`Main thread`
99,99 %	56477,0 ms	0 ms	`main`
99,99 %	56458,9 ms	6402 ms	`RaysPropagation`
29,8 %	16861 ms	16861 ms	`SnellFresnel`
28,9 %	16343,8 ms	514,1 ms	`CalculateOutput`
14,9 %	8434,5 ms	7732,5 ms	`GenerateRay`
9,5 %	5369,5 ms	5369,5 ms	`CalculatePlane`
7,5 %	4269,3 ms	4269,3 ms	`CalculateSphere`
2,5 %	1415 ms	1404,3 ms	`GetThetaFromDistance`

Tabla 4.4: Resultados del *profiler*

En concreto, se observa una simulación consistente en 121 mililentes y 1024 leds con un nivel de muestreo de 32 niveles por coordenada angular (1024 puntos). De entre

todos los procesos, `RaysPropagation` destaca sobre los demás en duración. Recordemos que dicha rutina aloja tanto el preprocesado como la simulación propiamente dicha. Sin embargo, puede comprobarse que acto y seguido las rutinas con mayor peso son `SnellFresnel` y `CalculateOutput`, mientras que las rutinas `GetThetaFromDistance` o `GetPhiLimits` apenas repercuten sobre el resultado final. Por este motivo, se decidió paralelizar el bucle de simulación presentado en la Subsección 4.4.3 que comprende las rutinas que más tiempo demandan dentro del código secuencial.

Dicho bucle está compuesto, a su vez, por 4 bucles: ledes, mililentes, ángulo azimutal y ángulo de elevación. Los rayos, una vez creados, son independientes por lo que *a priori* la paralelización podría hacerse sobre cualquier bucle. No obstante, los dos bucles más internos, los angulares, dependen de datos calculados en los dos más externos. Por consiguiente, la decisión de paralelizar queda reducida a hacerlo por ledes o por mililentes. Debido al artefacto del clúster, el número de ledes, normalmente, es sensiblemente mayor al número de mililentes por lo que se decidió paralelizar el bucle más externo y distribuir las iteraciones entre el número de hilos. El número de hilos se decidió pasar por parámetro al código.

Ya que OpenMP considera como variables compartidas todas aquellas variables existentes antes de la paralelización, cada hilo deberá poseer la información relativa a su rayo de forma privada. Esta información son los ángulos θ y ϕ. Además, estos dependen de los diferenciales de ángulo, $\mathrm{d}\theta$ y $\mathrm{d}\phi$, y de las variables correspondientes al preprocesado en elevación: $s_{\text{mín}}$, $s_{\text{máx}}$, $\theta_{\text{mín}}$ y $\theta_{\text{máx}}$. Aunque pueda parecer un número excesivo de variables de tipo `private`, hay que entender que cada hilo se ocupa de un único rayo al mismo tiempo y que estos pueden pertenecer a pares led-mililente diferentes, con lo que hay que preservar los valores geométricos. Los índices de los bucles también se han declarado como variables privadas para poder acceder correctamente al *array* de rayos.

Con todo, el código se ha paralelizado utilizando la siguiente directiva sobre el bucle externo:

```
#pragma omp parallel for private(i, j, k, l, diffP, phi, solMin,
solMax, thetaMin, thetaMax, theta, diffT)
```

4.7. Código GPU

La versión para **CUDA®** del código secuencial se basa en añadir a las rutinas secuenciales que se llamen desde dentro del bucle paralelizado en OpenMP el atributo `__device__` para que se puedan ejecutar en la GPU. La única excepción es la función `dot_product` a la que se le ha añadido los atributos `__host__` y `__device__`. De la misma forma, se ha declarado dos *kernels*: uno que engloba a todos los rayos del bucle, `cudaRaysPropagation`, y otro para los rayos directos, `cudaDirectRays`.

En cuanto a la estructura de bloques e hilos a emplear, se decidió fijar el número de hilos por bloque mediante un parámetro del código y, en función de su valor, calcular el número de bloques necesario. Tanto el *grid* de bloques como los bloques de hilos son unidimensionales, acordes al vector de rayos que se emplea. El número de modificaciones

a realizar fue escaso debido a que se tuvieron en cuenta desde el código secuencial: reducción de *branches* mediante operadores ternarios donde fuese posible, aplanamiento de las estructuras de datos, secuencialidad de la memoria, etc.

Capítulo 5

Resultados

Este capítulo presenta los resultados obtenidos a nivel de simulador en cuanto a patrones de radiación sobre el plano imagen, así como las aceleraciones y eficiencias obtenidas fruto de la paralelización. De la misma forma, se introduce un pequeño estudio del compromiso entre el tamaño del clúster y la precisión de la simulación.

5.1. Equipo empleado

Aquí se presenta el equipo empleado a la hora de obtener los resultados. Para todas las implementaciones, se ha hecho uso de una máquina disponible en el DTFC. Las características principales (extraídas con `lscpu` y `deviceQuery`) son las de la Tabla 5.1.

5.2. Estudio de la precisión

Para el estudio de la precisión se ha partido de la base que las mililentes ya están elegidas por lo que todos los parámetros que hacen alusión a las mismas se mantendrán estáticos. Esta decisión nos deja la libertad de elegir la distancia de los ledes a las mililentes, al igual que la de las mililentes al plano de impacto, número de ledes y mililentes, índice del emisor lambertiano y potencia, y por último, los niveles de muestreo en las coordenadas angulares.

De entre ellas se ha descartado la distancia de las mililentes al plano de impacto puesto que es irrelevante de cara a la precisión, de la misma forma, la potencia de las fuentes no influye en el resultado final. Además, teniendo en cuenta que las dimensiones de trabajo son inferiores al milímetro se ha decidido emplear únicamente un led por mililente ya que las conclusiones que se extraigan son igualmente válidas al aumentar el número de ledes porque la separación entre ellos es mínima.

Con todo, el escenario base para todos los casos es el siguiente:

- Distancia a las mililentes: 1 mm.

Intel® Core™ i7-4770K	
Arquitectura	x86-64
Núcleos	4
Hilos por núcleo	2
Reloj	3530,761 MHz
Caché L1d	32 kbytes
Caché L1i	32 kbytes
Caché L2	256 kbytes
Caché L3	8192 kbytes
NVIDIA GeForce® GTX 680	
Versión del controlador	7.0
Capacidad de cómputo	3.0
Memoria global	2048 Mbytes
Núcleos	1536
Reloj	1202 MHz
Ancho de bus	256 bits GDDR5
Ancho de banda máximo	192,2 GBps
Caché L2	512 kbytes
Memoria	
Disco duro de estado sólido	512 Gbytes
Disco duro mecánico	1 Tbytes
RAM	16 Gbytes DD3 @ 1333 MHz

Tabla 5.1: Características del equipo empleado

- Distancia al plano de impacto: 3 m.
- Índice de refracción del medio: 1 (aire).
- Índice de refracción de las mililentes: 1,46 (cuarzo).
- Número de ledes: 1 por mililente.
- Separación de las mililentes: 1,5 mm.
- Diámetro de las mililentes: 1,46 mm.
- Espesor de las mililentes: 1,24 mm.
- Tamaño de la protuberancia (cúpula): 114,73 μm.
- Radio de curvatura de las mililentes: 2,38 mm.
- Índice del emisor lambertiano: 1.
- Potencia total de un emisor: 1 W.
- Número de niveles en azimut y elevación: 2^5.

Por consiguiente, el estudio de la precisión se hará sobre tres parámetros: índice del emisor lambertiano, m; distancia de los ledes a las mililentes, z_1; y número de niveles en azimut y elevación por pareja de led-mililente, 2^P y 2^T, respectivamente. En cada caso y barriendo el tamaño del clúster (el número de mililentes) se obtiene la potencia recibida en el plano impacto. Debido a que el clúster es una aproximación de la simulación de una pantalla completa, se presupone que el error cometido por dicha aproximación se ve reducido en tanto en cuanto el número de mililentes aumente y se parezca más a la situación real. Se ha definido, por tanto, la precisión como $1 -$ error relativo, siendo el error relativo la diferencia entre dos simulaciones cuyo tamaño del clúster es consecutivo.

En primer lugar, se presentan los resultados al variar el patrón de radiación del led. Recordemos que un valor mayor de m implica un menor ángulo mitad (ángulo al cual la irradiancia de la fuente ha decaído hasta la mitad del valor máximo que se encuentra en el eje axial). De forma intuitiva, si el patrón de radiación es relativamente ancho necesita un número de lentes mayor. En la Figura 5.1 pueden verse los resultados para ángulos mitad de $\pm 30°(m = 4,82)$, $\pm 40°(m = 2,60)$, $\pm 50°(m = 1,57)$ y $\pm 60°(m = 1)$.

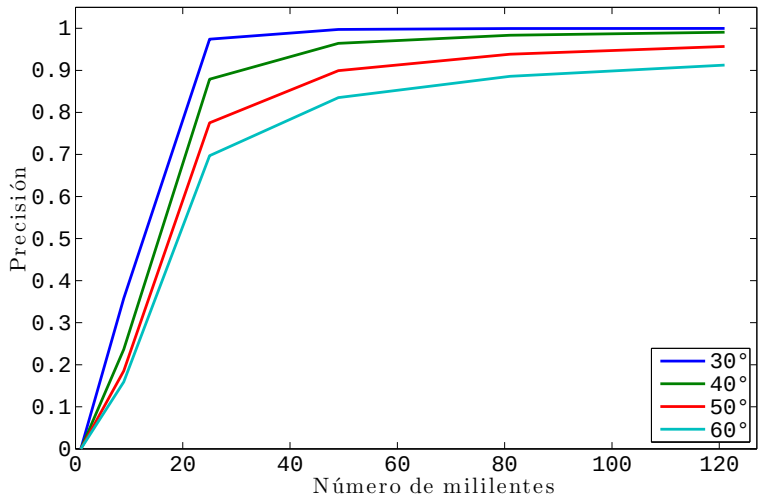

Figura 5.1: Precisión en función del ángulo mitad (índice lambertiano)

De la Figura 5.1 se comprueba que, en efecto, según m aumenta, el tamaño del clúster necesario para que la simulación sea precisa disminuye. Por ejemplo, si empleamos un patrón lambertiano puro ($\pm 60°$) para un clúster de tamaño 5 (121 mililentes) la precisión de la medida es de tan solo del 91 %, mientras que para un patrón con ángulo mitad de $\pm 40°$se consiguen resultados similares con simplemente 49 mililentes (96,42 %). Se intentó obtener una expresión analítica para predecir el tamaño del clúster necesario en función del patrón de radiación de la fuente, pero ésta depende del rayo refractado y habría que obtenerla por medios numéricos. Esto implica que a la hora de simular bastaría con hacer un pequeño estudio previo para conocer el tamaño del clúster necesario para una precisión dada.

De la misma forma, al variar la separación entre los ledes y las mililentes, el ángulo sólido que cubre una única fuente aumenta a medida que la distancia se incrementa y se necesita un mayor número de mililentes para que la simulación sea precisa. La Figura 5.2 ilustra este fenómeno donde sobre el escenario base se ha variado la distancia desde

0 mm hasta 3 mm en pasos de 1 mm. En el caso extremo, cuando la separación es nula, el valor de la medida no cambia, de ahí que la precisión de la medida no cambie según se aumenta el número de elementos. Por el contrario, en el resto de casos, según se aumenta la distancia, para conseguir una mayor precisión se necesitan un alto número de elementos. En otro caso extremo, 3 mm, la precisión obtenida para 121 lentes no sobrepasa el 84,25 %.

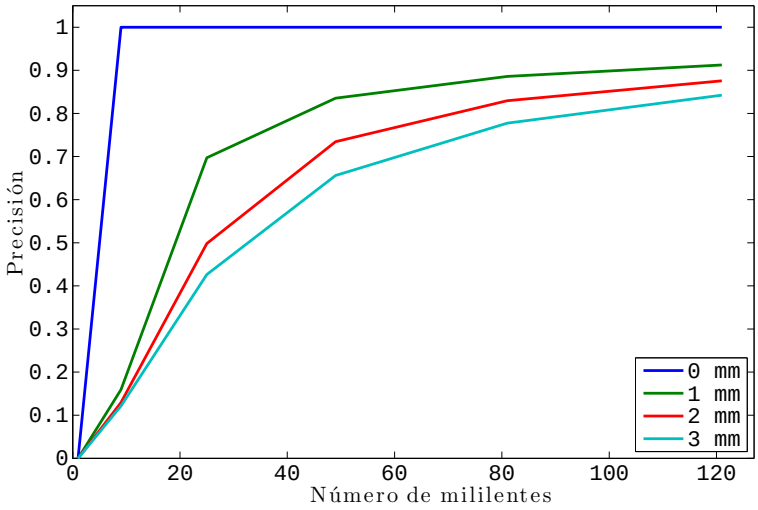

Figura 5.2: Precisión en función de la separación de la fuente

Por último, se ha simulado el escenario base cambiando el nivel de muestreo de las coordenadas angulares, esto es, el número total de rayos a simular. Este número se obtiene de multiplicar el número de mililentes del clúster por el número de ledes por mililente, además de por el número de niveles (potencias de dos) más los rayos directos. Por ejemplo, para un clúster de tamaño 3 (49 mililentes) con una única fuente y 5 niveles en ambas coordenadas se genera 50177 rayos ($49 \cdot 1 \cdot 32 \cdot 32 + 1$). En la Figura 5.3 puede comprobarse que un aumento en la resolución angular no implica una medida más precisa, no así el aumento de elementos del clúster (que coincide con el escenario base con $m = 1$ y $z_1 = 1$ mm). Los niveles de muestreo elegidos han sido 16, 32, 64 y 128.

Con este pequeño estudio, se puede concluir que en función del escenario a simular es crucial elegir el número de elementos del clúster para que los resultados sean tan precisos cómo se desee. Esto añade un paso previo al flujo de trabajo presentado en la Figura 1.2, que desde el punto de vista del autor, no añade una complejidad excesiva al modelo propuesto ya que de forma intuitiva, con este somero estudio, se entiende perfectamente el compromiso a seguir entre tamaño del clúster y los dos parámetros clave: índice del emisor lambertiano y distancia a las mililentes. El muestreo espacial añade precisión desde el punto de vista del funcional puesto que aumenta el soporte sobre el que obtenerlo, pero no sobre el resultado de la simulación como tal.

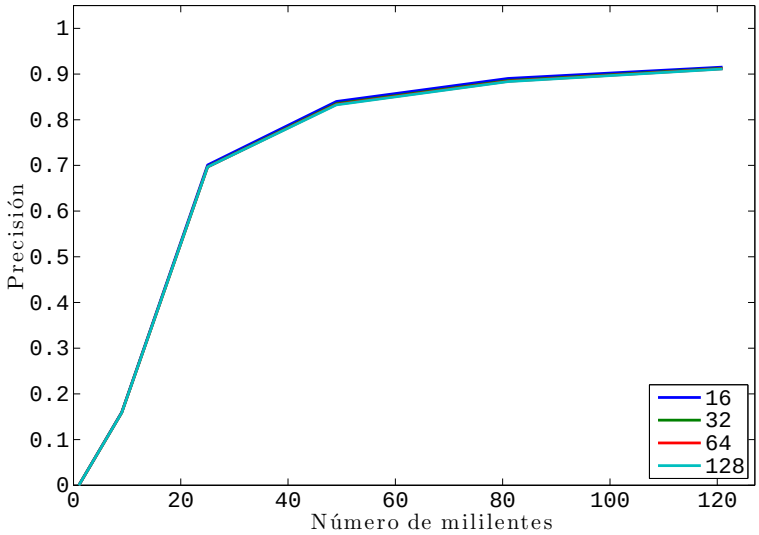

Figura 5.3: Precisión en función del nivel de muestreo angular

5.3. Resultados de la paralelización

En esta sección se presentan las aceleraciones y eficiencias resultado de la paralelización del código. Primero, se presentan los resultados relativos a la paralelización con memoria compartida donde se ha calculado la aceleración, así como la eficiencia de dicha paralelización. Por el contrario, para la paralelización sobre GPU sólo se introducen los resultados relativos a aceleración puesto que la eficiencia hace alusión al número de procesadores hardware y no procede para este caso. Todos los tiempos, salvo que se indique lo contrario, se han medido como *wall-clock time* así que los tiempos de transferencia de datos entre la CPU y la GPU están contemplados.

En cualquier caso, se ha recogido un total de 43 escenarios distintos variando el número de rayos a partir del escenario base, variando la configuración física del escenario mediante el número de mililentes y ledes además del muestreo espacial. En ningún caso, el tiempo total de simulación del código secuencial supera los 5 segundos en simple precisión y los 4 segundos en doble precisión (menos rayos totales). Esto puede dar lugar a pensar por qué paralelizar cuando el tiempo de simulación es tan breve, no obstante, el sistema a optimizar presentado en la Figura 1.2 es un sistema iterativo, así que cualquier ganancia es bienvenida.

5.3.1. OpenMP

A continuación se presenta los resultados obtenidos al paralelizar el código en OpenMP sin especificar ninguna estrategia de planificación, por consiguiente, cada hilo ejecutará un total de n/h iteraciones; siendo n el número de iteraciones total (rayos) y h el número de hilos. En la Figura 5.4 puede comprobarse los resultados obtenidos para 2, 4, 8 y 16 hilos, respectivamente.

El procesador empleado posee 4 núcleos físicos y otros tantos lógicos. Esta es la

razón por la que se obtienen aceleraciones lineales para 2 y 4 hilos, mientras que para 8 y 16 la tasa disminuye. Destacar que para todos los hilos existe una serie de casos donde la aceleración es ligeramente inferior a la unidad (resaltados en rojo), esto es debido a que, a pesar de que el número de rayos aumenta, en el escenario, únicamente hay un led. Si recordamos de la Sección 4.6, la opción elegida fue paralelizar por fuentes por lo que no existe ninguna ventaja frente al caso secuencial.

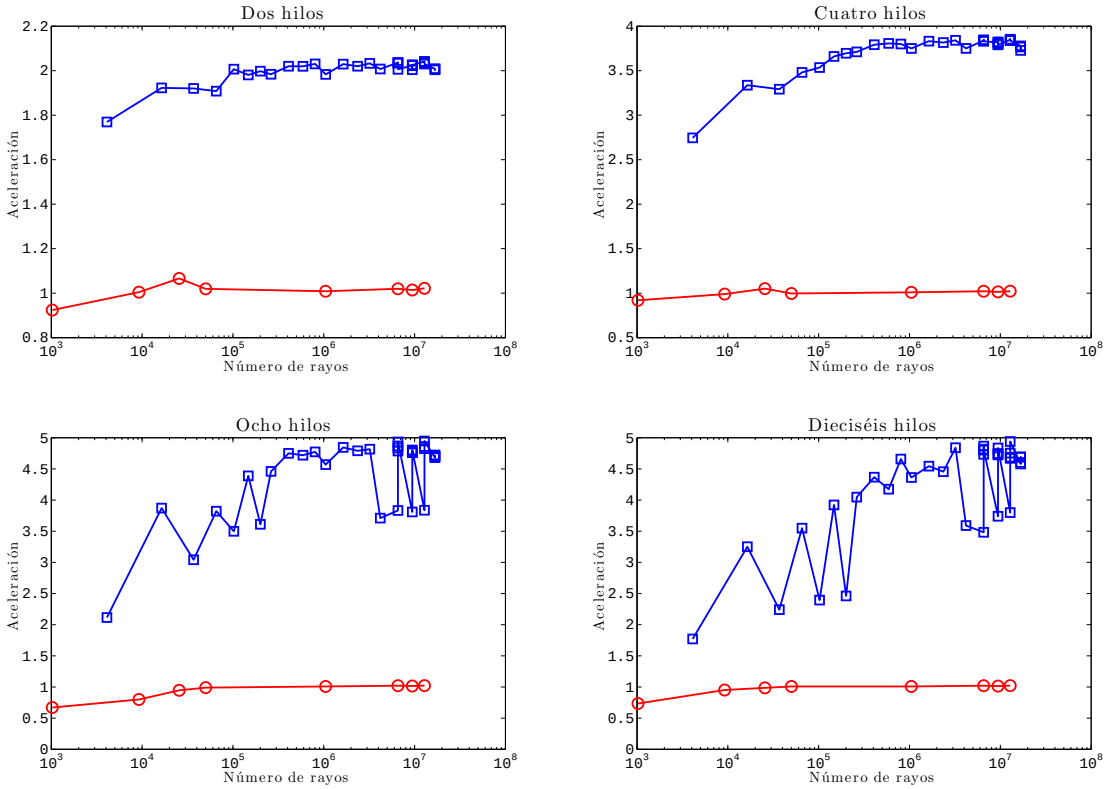

Figura 5.4: Aceleraciones para memoria compartida

En la Figura 5.5 se presenta la eficiencia para los distintos casos, se comprueba a simple vista que la eficiencia no varía apenas a partir de 4 hilos puesto que no hay más procesadores y el tiempo es muy similar. Hay que recordar que se están empleando núcleos lógicos (software) que emulan el comportamiento de uno real y por lo tanto, son resultados subóptimos.

De los resultados obtenidos, se observa que la paralelización es bastante eficiente puesto que la eficiencia para los casos de 2 y 4 es cercana a la unidad. Si se hubiese simulado en un procesador con mayor número de núcleos, se podría haber comprobado que el comportamiento debe ser el mismo. De la misma forma, se ha obviado el estudio de otras planificaciones, estáticas o dinámicas, en el código por una sencilla razón: todos los rayos ejecutan las mismas operaciones. Esto es así porque se ha empleado un muestreo determinista para cada par led-mililente y además porque un rayo a pesar de no refractarse, a efectos de cálculo, sí se tiene en cuenta por lo que se sigue operando sobre él.

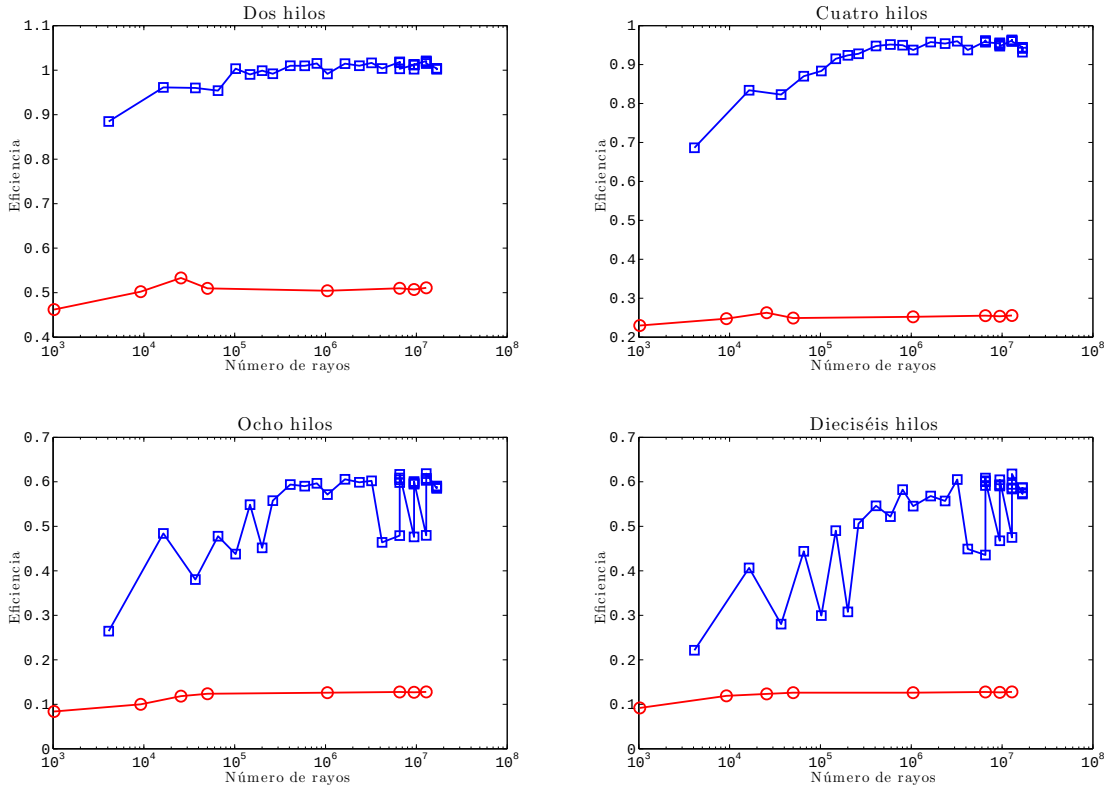

Figura 5.5: Eficiencias para memoria compartida

Doble precisión

Aprovechando que el código está parametrizado para trabajar tanto en coma flotante de simple y doble precisión, se repitieron las simulaciones utilizando en este caso aritmética de doble precisión. Los resultados pueden observarse en las Figuras 5.6 y 5.7. En estos casos, salvo para 2 hilos, las aceleraciones no son lineales. Además, se observa el mismo comportamiento errático en 8 y 16 hilos cuando se supera el número de *cores* físicos disponibles en el procesador.

Llegados a este punto, cabe preguntarse si el aumento de la precisión es asumible teniendo en cuenta la pérdida de rendimiento. Para ello se compara en las Tabla 5.2 el error cuadrático medio entre el prototipo **MATLAB®** y el código multihilo para 9 mililentes y 4 ledes. Los guarismos de error que se manejan para simple precisión están en el orden de la millonésima, mientras que para doble precisión estos disminuyen seis órdenes de magnitud aproximadamente. Teniendo en cuenta que la versión en simple precisión es más que suficiente para el objetivo propuesto y que la versión en doble precisión no mantiene el comportamiento lineal con el número de hilos, desde el punto de vista del autor, no compensa aumentar la precisión en ningún caso.

5.3.2. CUDA®

En la versión **CUDA®**, el número de bloques se calcula a partir del número de rayos a simular y el número de hilos por bloque, así que se ha barrido todos los escenarios

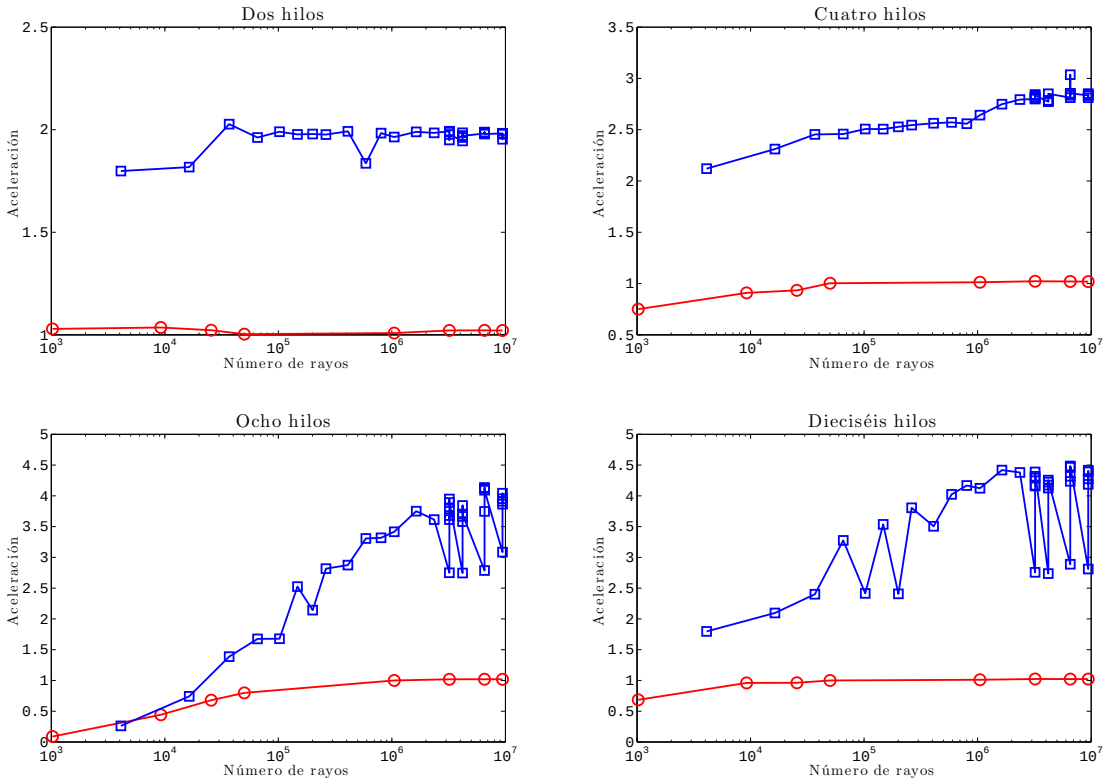

Figura 5.6: Aceleraciones para memoria compartida en doble precisión

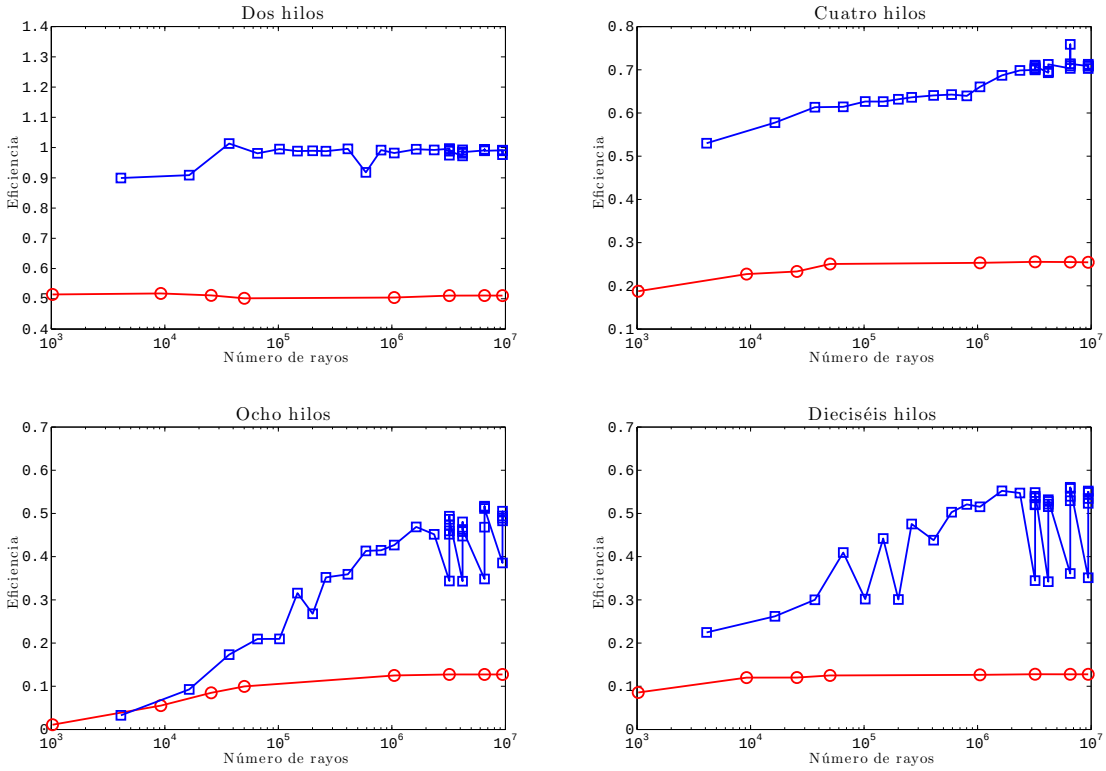

Figura 5.7: Eficiencias para memoria compartida en doble precisión

Error cuadrático medio en simple precisión		
Rayos generados	Distancia	0
	Potencia	2,9370e-07
	Posición	[5,4210e-20 5,4210e-20 0]
	Vector director	[2,7908e-07 2,7784e-07 3,0087e-07]
Rayos refractados	Distancia	2,8972e-09
	Potencia	3,2028e-07
	Posición	[2,1771e-09 2,1780e-09 2,8223e-09]
	Vector director	[2,8213e-07 2,8156e-07 3,6580e-07]
Rayos impactados	Distancia	7,2630e-06
	Potencia	3,2028e-07
	Posición	[5,0440e-06 5,0761e-06 1,5399e-16]
	Vector director	[2,8213e-07 2,8156e-07 3,6580e-07]
Error cuadrático medio en doble precisión		
Rayos generados	Distancia	0
	Potencia	5,1938e-15
	Posición	[5,4210e-20 5,4210e-20 0]
	Vector director	[2,9964e-14 2,9972e-14 1,6284e-14]
Rayos refractados	Distancia	5,9922e-17
	Potencia	3,6580e-14
	Posición	[7,9599e-17 7,9637e-17 2,6966e-17]
	Vector director	[1,7801e-14 1,7802e-14 4,4949e-14]
Rayos impactados	Distancia	1,4574e-12
	Potencia	3,6580e-14
	Posición	[1,0794e-12 1,0801e-12 1,5399e-16]
	Vector director	[1,7801e-14 1,7802e-14 4,4949e-14]

Tabla 5.2: Error cuadrático medio según el tipo de coma flotante empleado

para un número de hilos tal que 2^h con $h = 0, \ldots, 10$. Para representar mejor los datos se ha decidido dividir los resultados en las Figuras 5.8 y 5.9.

En la Figura 5.8 se muestra los resultados desde 1 hasta 32 hilos por bloque. En todos los casos se consigue aceleraciones por encima de la unidad a partir de 4 millones de rayos. Además, el rendimiento aumenta según aumenta el número de hilos, lo que de forma implícita implica una reducción del número de bloques. Sin embargo, en la Figura 5.9 puede comprobarse como esto deja de ser cierto a partir de 64 hilos. En efecto, para 64 hilos el rendimiento disminuye, mientras que para el resto de casos vuelve a aumentar pero sin mejorar el caso de 32 hilos.

A la luz de los resultados, el mejor caso se da cuando se eligen 32 hilos por bloque para todos los escenarios considerados. No obstante, las aceleraciones obtenidas son pobres en comparación con el paradigma de memoria compartida. Para buscar una explicación a este hecho, se debe tener en cuenta el paso de memoria entre la GPU y la CPU para recolectar los resultados de la simulación. Los resultados de la simulación son 3 *arrays* de rayos (generados, refractados e impactados) que en muchos casos tienen varios millones de elementos hasta alcanzar prácticamente la capacidad total de memoria de la GPU. Para comprobarlo, se ha hecho uso del *profiler* que proporciona

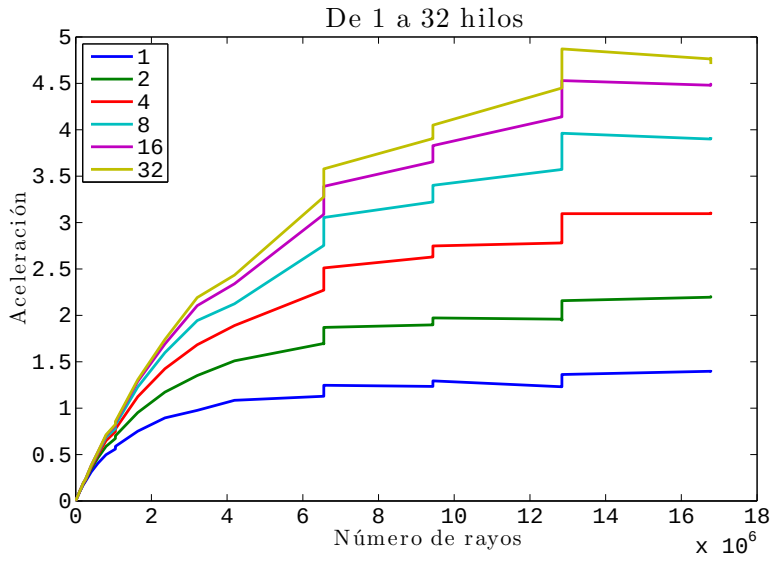

Figura 5.8: Aceleraciones para GPU de 1 a 32 hilos por bloque

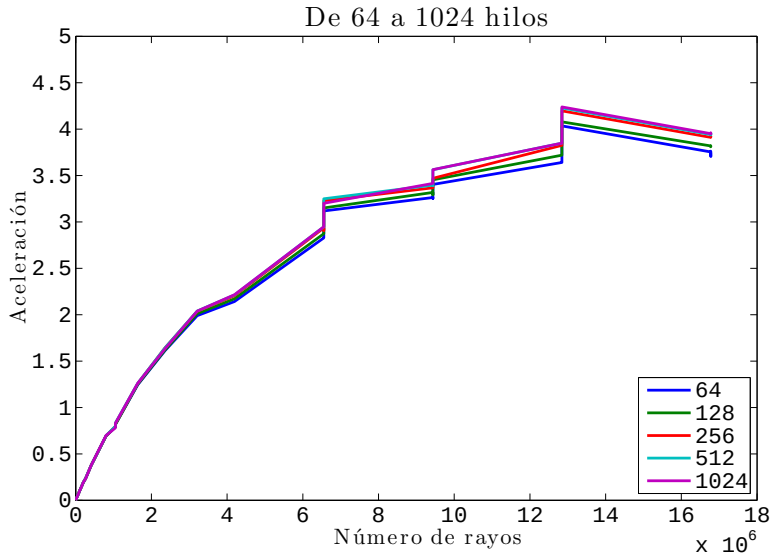

Figura 5.9: Aceleraciones para GPU de 64 a 1024 hilos por bloque

NVIDIA, `nvprof`, para 1 mililente y 1024 ledes. Los resultados de los *kernels* y `memcpy`, así como las llamadas principales a la API, se muestran en la Tabla 5.3.

Llamadas a los *kernels* y `memcpy`						
Tiempo (%)	**Tiempo**	**Llamadas**	**Media**	**Mínimo**	**Máximo**	**Nombre**
59,41 %	380,25 ms	1	380,25 ms	380,25 ms	380,25 ms	CUDA `memcpy` DtoH
40,59 %	259,79 ms	1	259,79 ms	259,79 ms	259,79 ms	`cudaRaysPropagation`
0,01 %	46,752 μs	1	46,752 μs	46,752 μs	46,752 μs	`cudaDirectRays`
0,00 %	4,7360 μs	5	947 ns	576 ns	1,4400 μs	CUDA `memcpy` HtoD
Llamadas a la API						
Tiempo (%)	**Tiempo**	**Llamadas**	**Media**	**Mínimo**	**Máximo**	**Nombre**
54,36 %	380,49 ms	6	63,414 ms	3,3810 μs	380,45 ms	`cudaMemcpy`
37,11 %	259,80 ms	1	259,80 ms	259,80 ms	259,80 ms	`cudaEventSynchronize`
5,04 %	35,262 ms	6	5,8770 ms	4,3220 μs	34,455 ms	`cudaMalloc`
3,30 %	23,097 ms	1	23,097 ms	23,097 μs	23,097 ms	`cudaDeviceReset`

Tabla 5.3: Resultados de `nvprof`

De hecho, si se mide la fracción de código paralelizada, el bucle de rayos, los resultados son bastante mejores. Hay que tener en cuenta que cuando serializa la llamada a un *kernel* alojado en la GPU, el *host* no queda bloqueado, es decir, puede seguir ejecutando código en paralelo al procesador gráfico, de forma que únicamente quedará bloqueado en caso de hacer una petición de lectura de memoria de tipo `memcpy` entre *device-host*. Este hecho genera un problema a la hora de realizar mediciones de tiempo, ya que con instrucciones de procesador comunes como `clock` no es posible medir los tiempos de cómputo en la GPU. Para ello, en [27] se describe cómo hacerlo correctamente empleando `cudaEventSynchronize`. La Figuras 5.10 recoge los resultados exclusivamente de la simulación.

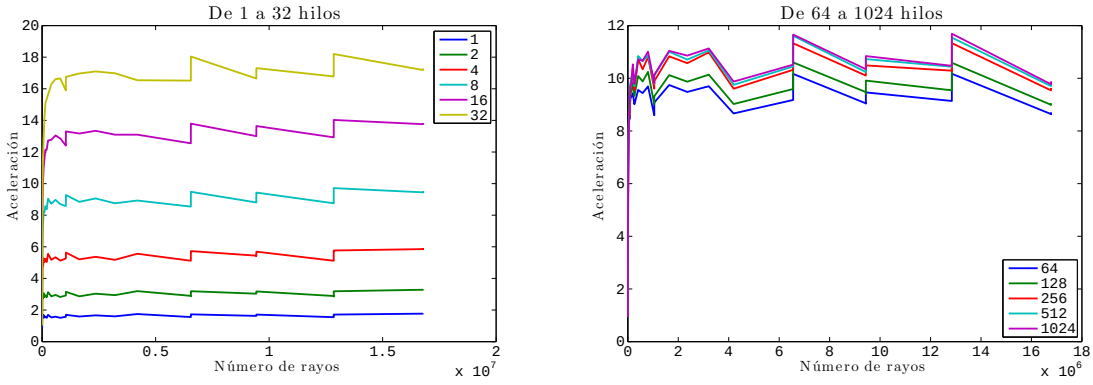

Figura 5.10: Aceleraciones para GPU contando sólo la simulación

Cuando se tiene en cuenta la simulación los resultados son prácticamente planos para todos los escenarios, sin importar la configuración geométrica escogida. De la misma forma, las aceleraciones son mayores a la unidad para todos los casos independientemente del número de rayos. Sin embargo, la tendencia observada en las aceleraciones globales se mantiene; el caso con 32 hilos es el que mejor se comporta, consiguiendo aceleraciones alrededor de 18 para ciertos casos. Para mostrar mejor este hecho, en la Figura 5.11 se comprueba la aceleración media para los distintos números de hilos donde se muestra de una forma más clara las zonas con aumento o disminución del rendimiento. Cambiando el código paralelo en **CUDA®** se podría hacer un análisis forzando el número de bloques pero no se creyó necesario.

De los resultados obtenidos, puede concluirse que el verdadero cuello de botella de la aplicación reside en el paso de memoria entre la GPU y la CPU. En [28] se explica el uso de memoria no paginada (*pinned*) para agilizar la transferencia por DMA. *Grosso modo* se sustituye la función `malloc`, propia de C, por `cudaMallocHost`. Los resultados lejos de mejorar empeoran como puede apreciarse en la Figura 5.12.

Esto es debido a que la transferencia de memoria se ha reducido a costa de aumentar el tiempo de reserva de memoria. Esto es fácilmente comprobable gracias a `nvprof` con la misma configuración: 1024 ledes y una sola mililente. En la Tabla 5.4 se muestra tanto los resultados de las principales funciones como los de la API.

La principal diferencia con la Tabla 5.3 es que el tiempo consumido por `cudaMemcpy` se ha visto reducido en un orden de magnitud, de cientos de milisegundos a decenas; pero, en cambio, las funciones `cudaHostAlloc` y `cudaFreeHost` consumen

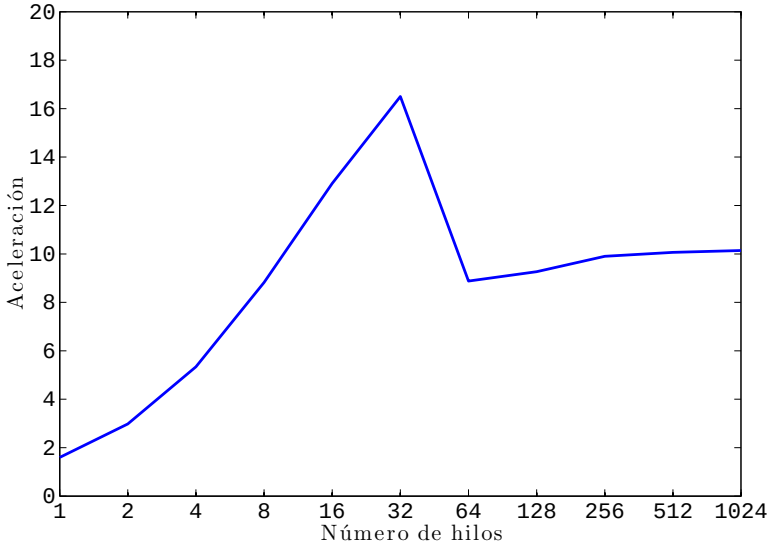

Figura 5.11: Aceleración media según el número de hilos por bloque

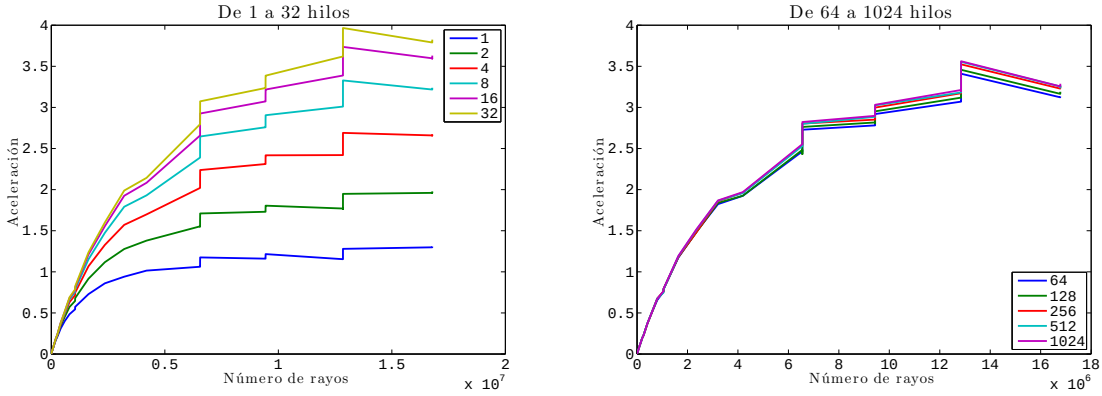

Figura 5.12: Aceleraciones para GPU con memoria no paginada

Llamadas a los *kernels* y `memcpy`						
Tiempo (%)	**Tiempo**	**Llamadas**	**Media**	**Mínimo**	**Máximo**	**Nombre**
64,92 %	262,37 ms	1	262,37 ms	262,37 ms	262,37 ms	`cudaRaysPropagation`
35,07 %	141,75 ms	1	141,75 ms	141,75 ms	141,75 ms	CUDA `memcpy` DtoH
0,01 %	46,879 μs	1	46,879 μs	46,879 μs	46,879 μs	`cudaDirectRays`
0,00 %	4,6720 μs	5	934 ns	576 ns	1,4080 μs	CUDA `memcpy` HtoD
Llamadas a la API						
Tiempo (%)	**Tiempo**	**Llamadas**	**Media**	**Mínimo**	**Máximo**	**Nombre**
32,20 %	304,40 ms	1	304,40 ms	304,40 ms	304,40 ms	`cudaHostAlloc`
27,75 %	262,37 ms	1	262,37 ms	262,37 ms	262,37 ms	`cudaEventSynchronize`
18,78 %	177,58 ms	1	177,58 ms	177,58 ms	177,58 ms	`cudaFreeHost`
15,00 %	141,83 ms	6	23,639 ms	3,0980 μs	141,80 ms	`cudaMalloc`
11,62 %	27,200 ms	6	4,5334 ms	3,9090 μs	27,165 ms	`cudaMemcpy`
9,68 %	22,647 ms	1	22,647 ms	22,647 ms	22,647 ms	`cudaDeviceReset`

Tabla 5.4: Resultados de `nvprof` para *pinned memory*

incluso más (en torno a 100 milisegundos). El tiempo de simulación de los *kernels*, `cudaRaysPropagation` y `cudaDirectRays`, no se ve afectado como cabría esperar puesto que sólo se ha intentado acelerar la transferencia de datos.

La última optimización que se aplicó para intentar ganar algo de tiempo en la transferencia se encuentra en [29]. La idea básica es solapar la transferencia de memoria con la ejecución del *kernel* a modo de *pipeline*. Por un lado, la GPU debe ser compatible: en nuestro caso lo es y cuenta con un *copy engine*. Por el otro, la ejecución de la transferencia de memoria así como el *kernel* debe hacerse en un *stream* distinto al de por defecto (*null stream*). Por último, la memoria debe ser reservada en formato *pinned*, lo cual ya está hecho del paso anterior. En la Figura 5.13 se muestra gráficamente lo que se intenta conseguir.

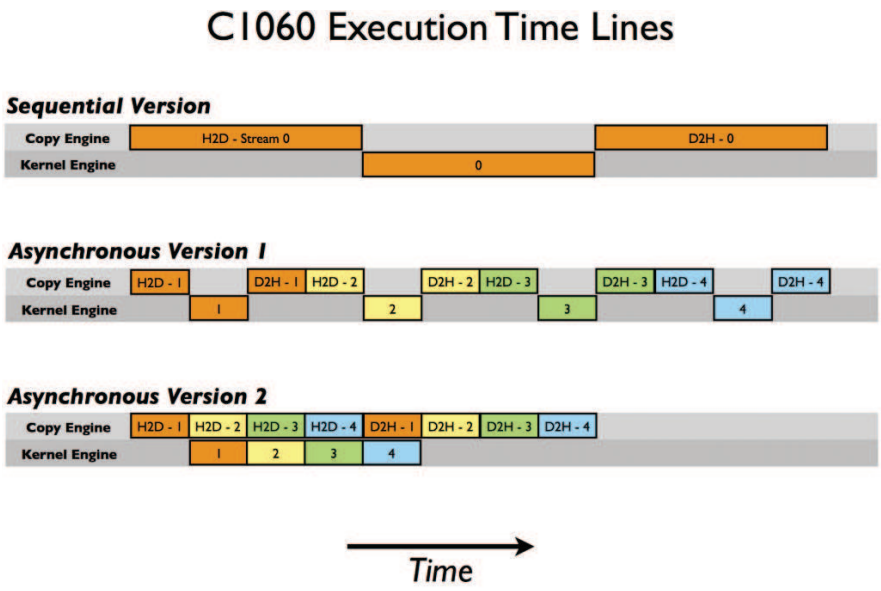

Figura 5.13: Ejecución solapada de *kernel* y transferencia de memoria

En GPU con capacidad de cómputo superior o igual a 3.5 ambas versiones son indistinguibles gracias al Hyper-Q, que permite ejecutar varios *kernels* de forma concurrente; sin embargo, la GPU sobre la que se realizaron las simulaciones posee una capacidad de cómputo 3.0 por lo que se ha implementado la versión 2 que es la más eficiente. Destacar que la tranferencia H2D, en nuestro caso, es prácticamente nula. Los resultados (ver Figura 5.14), de nuevo, no son destacables puesto que, para 32 hilos, el mejor caso, estamos dividiendo el tiempo de transferencia y de ejecución pero las funciones `cudaHostAlloc` y `cudaFreeHost` no han cambiado como se comprueba, a su vez, en la Tabla 5.5 para el caso anteriormente considerado.

Llamadas a los *kernels* y `memcpy`						
Tiempo (%)	**Tiempo**	**Llamadas**	**Media**	**Mínimo**	**Máximo**	**Nombre**
73,48 %	392,88 ms	2	196,44 ms	127,02 ms	265,86 ms	`cudaRaysPropagation`
26,51 %	141,75 ms	3	47,251 ms	9,7920 μs	70,896 ms	CUDA `memcpy` DtoH
0,01 %	46,847 μs	1	46,847 μs	46,847 μs	46,847 μs	`cudaDirectRays`
0,00 %	4,7680 μs	5	953 ns	544 ns	1,4720 μs	CUDA `memcpy` HtoD
Llamadas a la API						
Tiempo (%)	**Tiempo**	**Llamadas**	**Media**	**Mínimo**	**Máximo**	**Nombre**
46,22 %	463,58 ms	1	463,58 ms	463,58 ms	463,58 ms	`cudaEventSynchronize`
30,23 %	303,22 ms	1	303,22 ms	303,22 ms	303,22 ms	`cudaHostAlloc`
17,66 %	177,08 ms	1	177,08 ms	177,08 ms	177,58 ms	`cudaFreeHost`
3,47 %	34,769 ms	6	5,7949 ms	2,9660 μs	33,946 ms	`cudaMalloc`
2,28 %	22,860 ms	1	22,860 ms	22,860 ms	22,860 ms	`cudaDeviceReset`

Tabla 5.5: Resultados de `nvprof` para ejecución solapada de 2 flujos

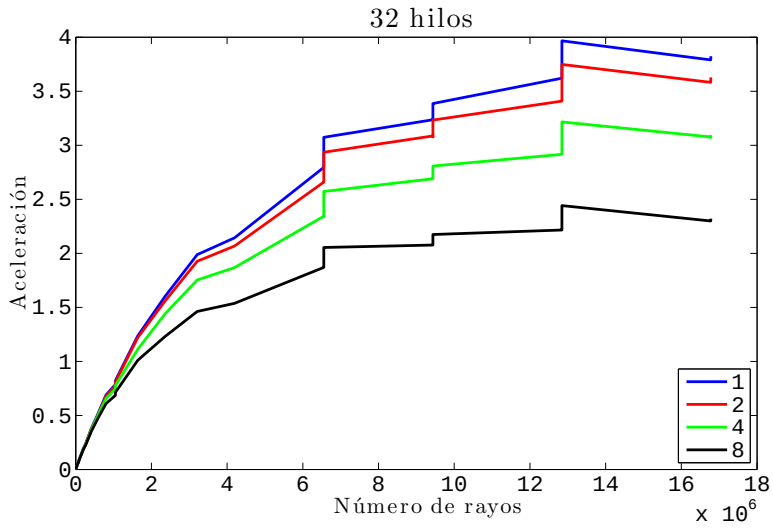

Figura 5.14: Aceleraciones con ejecución solapada para distinto número de flujos

El resto de funciones llamadas está dentro del orden del tiempo consumido en la Tabla 5.4, pero, al haber partido la ejecución de la simulación en dos, resulta que el *kernel* principal tarda algo más, yendo en detrimento del desempeño del sistema. Definitivamente, el cuello de botella de traerse los resultados a CPU penaliza sobremanera el rendimiento global de la implementación en GPU, por lo tanto, si se desease realizar el sintetizador con esta implementación se deberá considerar realizarlo en *kernel* para así aumentar el número de operaciones por rayo e intentar que la aplicación se convierta en intensiva en cómputo. A tenor de los resultados obtenidos con ambas implementaciones, memoria compartida y GPU, parece razonable pensar que la opción correcta para continuar el trabajo iniciado en este TFM es la de OpenMP.

Doble precisión

Se ha omitido los resultados relativos a las simulaciones sobre la arquitectura de GPU en doble precisión puesto que la discusión sobre la precisión ya se hizo para el paradigma de multihilo y además, al igual que entonces, el rendimiento es inferior.

5.4. Resultados del simulador

En todo momento, se ha hablado de la simulación de un clúster de mililentes pero no de una pantalla completa, esto es, recordemos, debido al principio de superposición subyacente al escenario planteado. Aunque con esto es más que suficiente para obtener el funcional descrito en la Ecuación 1.1, en esta etapa temprana del trabajo se creyó adecuado realizar una recreación de una pantalla completa sobre el plano imagen. Es por ello que se realizó un código auxiliar secuencial denominado `render` que, como su nombre indica, se encarga de *renderizar* en escala de grises la luz proyectada sobre el plano imagen.

El código recibe, por un lado, los resultados del código paralelizado en la memoria

más el tamaño de la pantalla en píxeles así como el área de integración en el plano imagen. Por otro lado, necesita una matriz de conmutación que indique la intensidad de cada píxel como un parámetro comprendido entre 0 y 1. De esta forma, se permite un cierto grado de flexibilidad, los mismos que se tendrían en el caso de un sintetizador. No se ha hecho alusión a este código en el resto de la memoria puesto que se trata de algo totalmente accesorio que únicamente sirve para completar los resultados obtenidos, parecido a los *scripts* que se han desarrollado para tratar los datos.

Para ilustrar esto, se ha elegido un escenario con los siguientes datos:

- Distancia a las mililentes: 1 mm.
- Distancia al plano de impacto: 3 m.
- Índice de refracción del medio: 1 (aire).
- Índice de refracción de las mililentes: 1,46 (cuarzo).
- Número de ledes: 16 por mililente.
- Número de mililentes del clúster: 49
- Separación de las mililentes: 1,5 mm.
- Diámetro de las mililentes: 1,46 mm.
- Espesor de las mililentes: 1,24 mm.
- Tamaño de la protuberancia (cúpula): 114,73 μm.
- Radio de curvatura de las mililentes: 2,38 mm.
- Índice del emisor lambertiano: 1.
- Potencia total de un emisor: 1 W.
- Número de niveles en azimut y elevación: 2^5.
- Dimensión del *array*: 128×128.
- Área de integración: 1 cm^2.
- Matriz de conmutación: todos encendidos.

El resultado de simular la pantalla completa sobre el plano de impacto se puede ver en la Figura 5.15. Se ha decidido generar una imagen en escala de grises para dar una idea de donde recae la energía. El código, `render`, toma a partir de los bordes físicos del *array* un ángulo arbitrario, 60°, e integra donde está la mayor parte de la energía. En muchos casos, la ausencia de rayos en una determinada zona implica que esa zona es oscura y no se debe en ningún caso a un problema de muestreo.

Hay que destacar que debido al tamaño del *array*, 128×128, y del número de mililentes por array, 64, el número total de mililentes que hacen falta son 16, formando

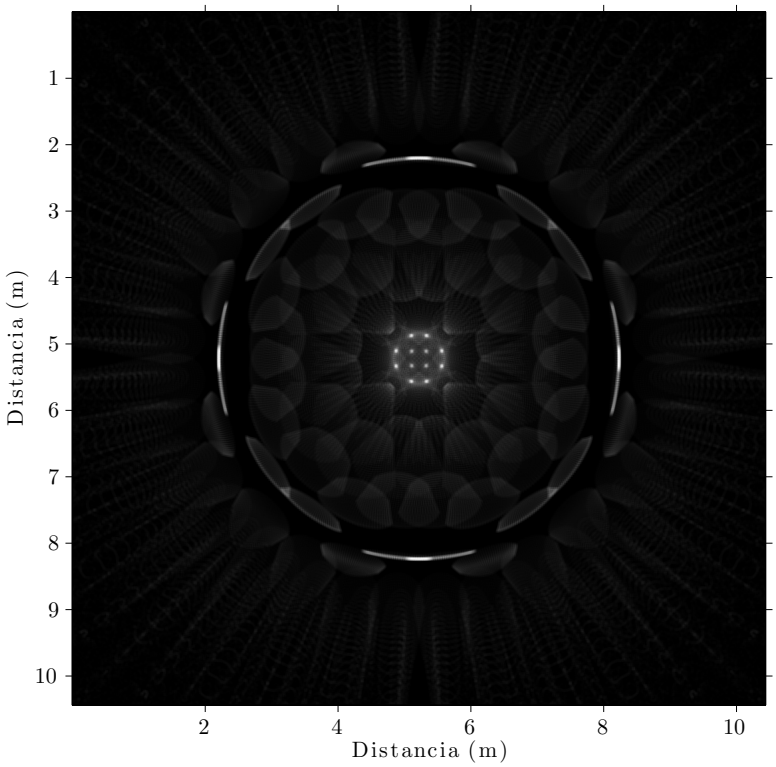

Figura 5.15: Renderizado de un *array* de 128 × 128

una matriz de 4 × 4. Además, el tamaño de cada led es una fracción del *pitch* de las mililentes o lo que es lo mismo: $1,5$ mm/32 ledes = $46,875$ μm. Estas dimensiones unidas a que el área de integración es de un centímetro cuadrado provocan que el resultado obtenido se parezca de forma notable a la simulación de un único clúster como se presenta en la Figura 5.16.

Para concluir, se modificó el radio de curvatura de las mililentes a la mitad. Esto implica además que para mantener el resto de parámetros inalterados hay que recalcular el tamaño de la protuberancia (casquete) esférico. Sin entrar en los valores concretos, de forma intuitiva, al reducir el radio de curvatura de las mililentes implica que la esfera es menor por lo que el círculo que forma con la superficie, que se emplea en el preprocesado, también es menor y por ello, menos rayos saldrán por las mililentes (todo lo que no entra por la cúpula es absorbido por la máscara). El resultado se puede ver en la Figura 5.17.

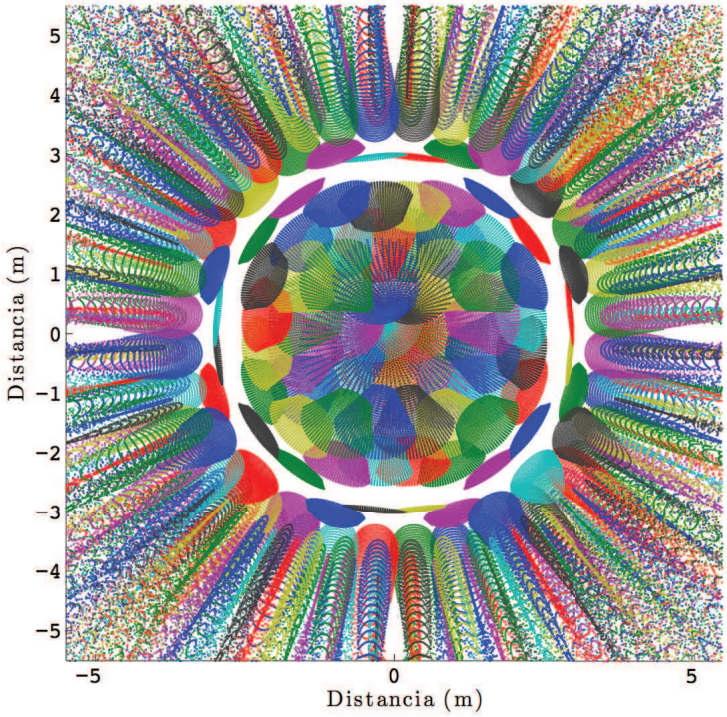

Figura 5.16: Clúster de 49 mililentes y 64 ledes

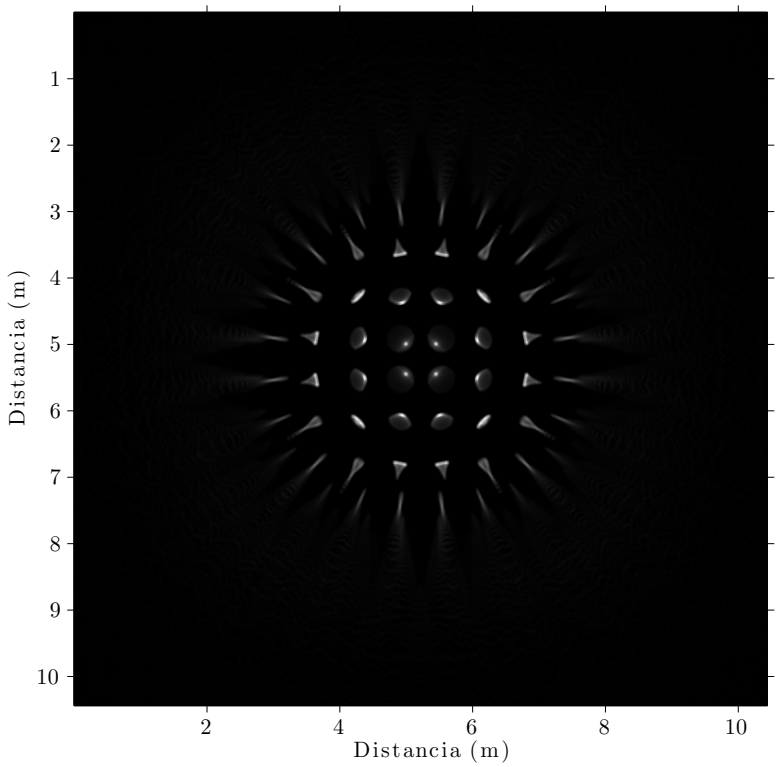

Figura 5.17: Renderizado de un *array* de 128 × 128 modificado

Capítulo 6

Conclusiones

En este capítulo final se exponen las conclusiones derivadas de los resultados obtenidos durante el TFM. En el presente trabajo se ha desarrollado un simulador óptico basado en trazado de rayos para una situación concreta a estudiar. Además, se ha paralelizado en diferentes arquitecturas paralelas para hacer viable su uso en un sistema de optimización mayor donde se requiera realizar repetidas simulaciones.

Si comparamos las dos estrategias de paralelización seguidas, OpenMP presenta eficiencias prácticamente unitarias cuando el procesador sobre el que se realizan las simulaciones cuenta con núcleos físicos disponibles; no obstante, si se sigue aumentando el número de hilos se continúan obteniendo aceleraciones pero no lineales. Por el contrario, **CUDA®** presenta aceleraciones muy pobres debido a la altísima cantidad de resultados que se transfieren al finalizar la simulación.

Respecto a los resultados de la simulación, en primer lugar, se procederá a validar los resultados obtenidos con el grupo CAFADIS antes del cálculo del funcional y posterior sistema de optimización, puesto que no se poseen resultados empíricos. Se ha observado que la agrupación de lentes en clústeres permite acelerar la simulación y además dar una idea muy aproximada del resultado final de la misma sin aplicar superposición.

Como aspectos a mejorar, en cuanto a la paralelización del algoritmo, cuando un rayo sufre de reflexión total interna, su potencia se vuelve nula y se sigue operando con él. Si se desechasen estos rayos se podría aumentar la aceleración del algoritmo paralelo. Esto repercutiría, por ejemplo, en permitir emplear estrategias dinámicas en lugar de estáticas en OpenMP. Igualmente, pasar de una malla estática a una malla adaptativa redundaría en reducir el número de rayos sin sacrificar por ello información relevante para la síntesis.

Aunque en el trabajo sólo se han implementado lentes plano-convexas esféricas, el simulador se ha dejado parametrizado para permitir la inclusión de nuevos tipos de lentes, así como plano-convexas parabólicas, plano-cóncavas, etc. De la misma forma, el modelo escogido para simular incluye una máscara que impide el paso de la luz, en el caso de escoger un modelo que careciese de la misma, bastaría con eliminar el preprocesado incluido y generar los rayos en todo el patrón de radiación.

Para finalizar, la opción de OpenMP parece la adecuada para continuar con el

trabajo iniciado, no obstante, si el resto del sistema se realizase sobre la arquitectura de GPU, esta decisión podría cambiar. Esto es debido a que se eliminaría la necesidad de enviar una gran cantidad de datos a la CPU, se pasaría de enviar todos los rayos para obtener el funcional a enviar únicamente los coeficientes necesarios para generarlo. De esta forma, la aplicación pasaría a ser intensiva en cómputo y se conseguirían aceleraciones similares a las obtenidas cuando sólo se considera el fragmento de código paralelizado.

Parte II

Bibliografía

Bibliografía

[1] J. Rodriguez-Ramos, J. Marichal-Hernandez, J. Luke, J. Trujillo-Sevilla, M. Puga, M. Lopez, J. Fernandez-Valdivia, C. Dominguez-Conde, J. C. Sanluis, F. Rosa, V. Guadalupe, H. Quintero, C. Militello, L. Rodriguez-Ramos, R. Lopez, I. Montilla, and B. Femenia, "New developments at CAFADIS plenoptic camera," in *Information Optics (WIO), 2011 10th Euro-American Workshop on*, pp. 1–3, June 2011.

[2] V. Guerra, C. Suarez-Rodriguez, S. Rodriguez, R. Perez-Jimenez, and J. Rodriguez-Ramos, "Plenoptics for optical wireless sensor networks," in *Information Optics (WIO), 2013 12th Workshop on*, pp. 1–3, July 2013.

[3] *Creating a desired lighting pattern with an LED array*, vol. 7058, 2008.

[4] *Homogeneous LED-illumination using microlens arrays*, vol. 5942, 2005.

[5] "IEEE Standard for Local and Metropolitan Area Networks–Part 15.7: Short-Range Wireless Optical Communication Using Visible Light," *IEEE Std 802.15.7-2011*, pp. 1–309, Sept 2011.

[6] S. Rajagopal, R. Roberts, and S.-K. Lim, "IEEE 802.15.7 visible light communication: modulation schemes and dimming support," *Communications Magazine, IEEE*, vol. 50, pp. 72–82, March 2012.

[7] E. Monteiro and S. Hranilovic, "Constellation design for color-shift keying using interior point methods," in *Globecom Workshops (GC Wkshps), 2012 IEEE*, pp. 1224–1228, Dec 2012.

[8] R. Drost and B. Sadler, "Constellation design for color-shift keying using billiards algorithms," in *GLOBECOM Workshops (GC Wkshps), 2010 IEEE*, pp. 980–984, Dec 2010.

[9] J. Luna-Rivera, R. Perez-Jimenez, J. Rabadan-Borjes, J. Rufo-Torres, V. Guerra, and C. Suarez-Rodriguez, "Multiuser CSK scheme for indoor visible light communications," *Opt. Express*, vol. 22, pp. 24256–24267, Oct 2014.

[10] O. Gonzalez, R. Perez-Jimenez, S. Rodriguez, J. Rabadan, and A. Ayala, "OFDM over indoor wireless optical channel," *Optoelectronics, IEE Proceedings* -, vol. 152, pp. 199–204, Aug 2005.

[11] J. Armstrong, "OFDM for Optical Communications," *Lightwave Technology, Journal of*, vol. 27, pp. 189–204, Feb 2009.

[12] S. Dissanayake and J. Armstrong, "Comparison of ACO-OFDM, DCO-OFDM and ADO-OFDM in IM/DD Systems," *Lightwave Technology, Journal of*, vol. 31, pp. 1063–1072, April 2013.

[13] Y. Jun, "Modulation and demodulation apparatuses and methods for wired/wireless communication system," July 9 2009. US Patent App. 11/989,620.

[14] N. Fernando, Y. Hong, and E. Viterbo, "Flip-OFDM for optical wireless communications," in *Information Theory Workshop (ITW), 2011 IEEE*, pp. 5–9, Oct 2011.

[15] V. Guerra, C. Suarez-Rodriguez, O. El-Asmar, J. Rabadan, and R. Perez-Jimenez, "Pulse width modulated optical OFDM," in *IEEE ICC 2015 - First Workshop on Visible Light Communications and Networking (VLCN) (ICC'15 - Workshops 24)*, (London, United Kingdom), June 2015.

[16] S.-M. Kim and S.-M. Kim, "Performance improvement of visible light communications using optical beamforming," in *Ubiquitous and Future Networks (ICUFN), 2013 Fifth International Conference on*, pp. 362–365, July 2013.

[17] F. Yaras, H. Kang, and L. Onural, "State of the Art in Holographic Displays: A Survey," *Display Technology, Journal of*, vol. 6, pp. 443–454, Oct 2010.

[18] "IHS Technology." `https://technology.ihs.com/389045/`. Último acceso en junio de 2015.

[19] "DigInfo TV." `http://www.diginfo.tv/v/10-0155-r-en.php`. Último acceso en junio de 2015.

[20] "OpenMP." `http://openmp.org/wp/`. Último acceso en junio de 2015.

[21] "GeForce GTX 680 Whitepaper." `http://international.download.nvidia.com/webassets/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf`. Último acceso en junio de 2015.

[22] "**CUDA®**C Programming Guide." `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#simt-architecture`. Último acceso en junio de 2015.

[23] "**RED ONE®**." `http://www.red.com/products/red-one`. Último acceso en marzo de 2015.

[24] "Microlens Arrays." `http://www.thorlabs.de/newgrouppage9.cfm?objectgroup_id=2861`. Último acceso en marzo de 2015.

[25] C. Schlick, "An Inexpensive BRDF Model for Physically-based Rendering," *Computer Graphics Forum*, vol. 13, no. 3, pp. 233–246, 1994.

[26] B. De Greve, "Reflections and Refractions in Ray Tracing ," 2007.

[27] "**CUDA®**C Best Practices Guide." `http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf`. Último acceso en junio de 2015.

[28] "How to Optimize Data Transfers in CUDA C/C++." http://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/. Último acceso en junio de 2015.

[29] "How to Overlap Data Transfers in CUDA C/C++." http://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/. Último acceso en junio de 2015.