VNIVERSIDAD NACIONAL
AVTNOMA DE
MEXICO

Universidad Nacional Autonoma de Mexico

Posgrado en Ciencia e Ingenier a de la Computacion

Ambientes virtuales con GPU y sensores RGB-D

T e s i s

Que para optar por el grado de:
Maestro en ingenier a (computacion)

P r e s e n t a

Nicolas Rodr guez Carreon

Director de la tesis:
Dr. Jose David Flores Penaloza, Facultad de Ciencias

Cotutor:
Dr. Jesus Savage Carmona, Facultad de Ingenier a

Mexico, D.F. Enero 2016

Resumen

En este trabajo analizamos dos problemas. Primero, cómo construir una representación tridimensional del ambiente por el que se mueve un robot humanoide y, segundo, cómo simular lo que el sensor de profundidad montado en él capturaría en posiciones y orientaciones arbitrarias dentro de la representación creada. Tener una solución a esos problemas permitirá desarrollar algoritmos que mejoren la planeación de la ruta del robot, y dar esa solución fue nuestro objetivo.

El robot proporciona datos de color y profundidad de una escena mediante un sensor RGB-D, y esos datos, por su tamaño, pueden procesarse en paralelo. El procesamiento en paralelo se realizó utilizando Unidades de Procesamiento Gráfico (GPU) por su accesibilidad y uso en aumento. Diseñamos e implementamos un sistema que construye una representación a partir de las medidas de profundidad del sensor y de los datos que el robot proporciona sobre la posición y orientación. Después, simulamos, dentro de la representación creada, un sensor colocado arbitrariamente.

Dentro los resultados, encontramos que, para el sistema diseñado, el procesamiento con un sistema en paralelo es considerablemente más rápido que el secuencial. También, que depender de las mediciones de posición y orientación que da el robot produce construcciones con errores cuantiosos.

Concluimos que el procesamiento en paralelo es mejor opción que el secuencial en este problema. También, que la entrada al sistema debe ser solamente la secuencia temporal de medidas de profundidad. Como ejemplo de un algoritmo que toma esa entrada, analizamos, detalladamente, un sistema del estado actual.

Agradecimientos

Agradecerle por escrito a todas las personas maravillosas que he conocido en mi vida sería, además de difícil, lato. He decidido, por tanto, hacer una lista (necesariamente incompleta) de nombres de algunas personas a quienes les podría escribir páginas de memorias, elogios y agradecimientos.

Le doy gracias a mis padres, Socorro y Nicolás, por darme todo. Al mejor *minion*, mi hermana Sara. A mi primer y mejor amigo, Sergio y a su familia. A mi amplia familia y, especialmente, a mis tres abuelas. A mi colegiala, Mariana, y a su familia. A tres buenos amigos, Darío, Gonzalo y John. A amigos muy queridos y de gran confianza, en más o menos orden de aparición, Memo, Yul, Ofelia, Rosma, el *chamoy-power*, Maricarmen, Arturo, Olaguibert, Estrella, el *niupi*, Fernanda, Cassandra, Pinedo, Daniel, Luis, Zenón, Ana, Mariana, Pedro, Javier, Pamela, Eliu, Rafael, Andrés, Yehú, Ixé, Miguel, Melisa, Angélica, César, Jimena, Jaime, Cecilia, Dulce y ella (ella sabe por qué). A grandes profesores y amigos, Emmanuel, José Arturo, Martha, Érik, Alejandra, Luis M., y, mayormente, a Ocáriz. Y a Yngwie J. Malmsteen.

Gracias a mis tutores, los doctores Flores y Savage. A mis sinodales, los doctores Escalante, García, y Gastelum. Y a Lulú, Amalia y Cecilia, por su excelente trabajo en el posgrado.

Muchas gracias al Consejo Nacional de Ciencia y Tecnología (CONACYT) por la beca otorgada para mis estudios.

Le agradezco a DGAPA-UNAM el apoyo proporcionado para la realización de esta tesis a través del proyecto PAPIIT IG100915 Desarrollo de técnicas de la robótica aplicadas a las artes escénicas y visuales .

Gracias perpetuas a la Universidad Nacional Autónoma de México porque me permitió tener educación superior y hasta hacer un posgrado.

Índice general

Índice de guras

Índice de tablas

(A los poetas que vendrán.)
Hay que ser implacables.
(No tengan, pues, clemencia con mis errores.)
Nuestra debilidad les dará fuerza
y acertarán en donde fracasamos.
Pero una vez borrados
(si nos recuerdan)
ojalá piensen
en que la perfección
es para siempre ajena a todo intento humano.
J.E. Pacheco.

Cap´tulo 1

Introducción

El avance de la tecnología y el incremento de uso de las Unidades de Procesamiento Gráfico (*Graphics Processing Units*, en adelante, GPU), desde 2003, ha sido notable. Una búsqueda rápida en *IEEE Xplore* de la palabra clave GPU devuelve alrededor de 6500 resultados. Aunque originalmente su uso se limitaba, como el nombre indica, a gráficos por computadora, actualmente se utilizan, por la cantidad masiva de procesadores que contienen, y por la cantidad de operaciones de punto flotante por segundo que son capaces de realizar (fig. 1.1), como auxiliares del CPU en áreas como la simulación de procesos físicos y biológicos, procesamiento de señales, procesamiento de imágenes, medicina, finanzas, por nombrar algunas. A este uso extendido de las GPU se le conoce como cómputo de propósito general en la GPU (GPGPU, *General-Purpose Computing on Graphics Processing Units*), o también, como cómputo acelerado por GPU (*GPU-accelerated Computing*). En la actualidad, CUDA, de la compañía NVIDIA, es la principal plataforma para desarrollar aplicaciones que usan la GPU.

Desde su introducción al mercado, en 2010, los sensores RGB-D (Microsoft Kinect, por ejemplo) se convirtieron en dispositivos de costo accesible para obtener datos tridimensionales del ambiente. Estos sensores capturan, en cada pixel, la información de color —como lo haría una cámara digital— y la distancia de la cámara al área representada por el pixel. La información adquirida con estos dispositivos se ha utilizado, por ejemplo, como interfaz con la computadora, como herramienta de adquisición para reconstruir objetos y ambientes virtuales, y como sistema de visión por computadora para robots.

Esta tesis tiene como propósito desarrollar un sistema de cómputo heterogéneo (CPU-GPU) que sea capaz de construir una representación del entorno de un robot con un sensor RGB-D montado, utilizando la posición, orientación y mapas de profundidad provistos por el robot; y simular lo que el sensor de profundidad capturaría en posiciones y orientaciones

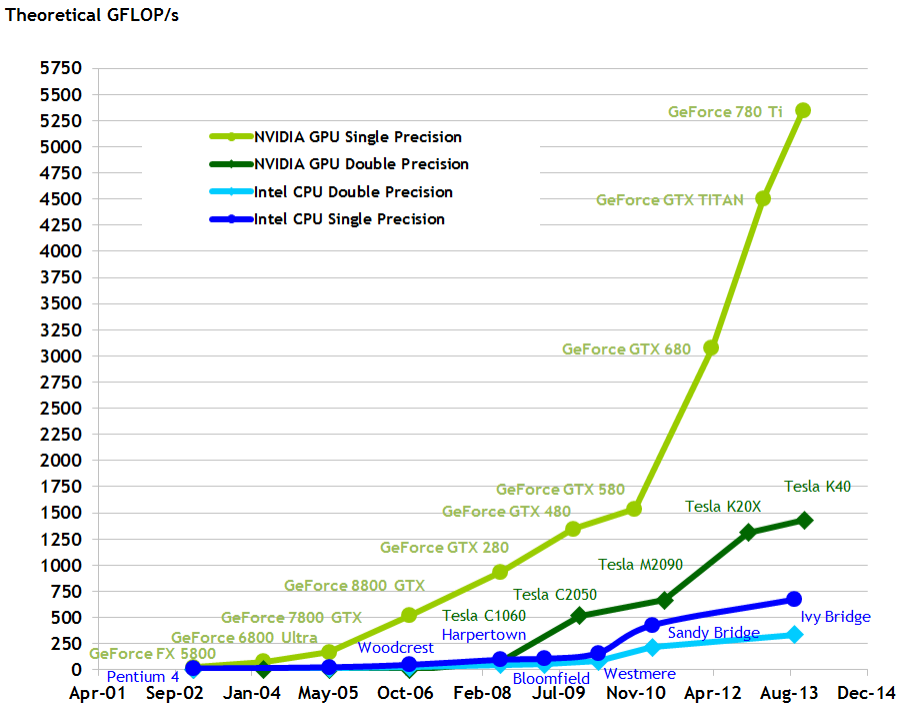


FIGURA 1.1: Operaciones de punto flotante por segundo (*Floating-point operations per second, FLOPS*) de una CPU contra una GPU a través de los años. [1]

arbitrarias dentro del ambiente virtual.

Tener una representación del ambiente, y poder simular dentro de este, es útil para grupos de robótica que utilizan un sensor RGB-D como método de adquisición de datos del entorno. Particularmente, para el grupo del Laboratorio de Biorobótica de la Facultad de Ingeniería, que está a cargo del Dr. Jesús Savage, para el desarrollo de algoritmos de planeación de rutas utilizando técnicas de Inteligencia Artificial.

Este trabajo se divide en siete capítulos que se describen, brevemente, a continuación.

El capítulo segundo de este trabajo es una introducción a la programación en GPU, y tiene la intención que pueda ser utilizada, por todos los interesados, como una primera aproximación al tema. Presenta una breve historia de la programación en GPU, algunos ejemplos de arquitecturas, conceptos de la plataforma CUDA, algunas consideraciones básicas para la optimización del rendimiento, y un ejemplo con una recomendación de bibliografía para instrucción propia.

El tercer capítulo ofrece un vistazo rápido a tres tecnologías diferentes para obtener un mapa de profundidad, haciendo énfasis en los patrones de luz estructurada y, particularmente, en el Kinect de Microsoft, por ser el dispositivo de más fácil acceso y el utilizado en esta tesis. En este capítulo también se presenta un modelo de cámara, los conceptos

de parámetros intrínsecos y extrínsecos de una cámara, y la utilidad de este modelo para obtener puntos en el espacio a partir de un mapa de profundidad.

Aunque el sistema presentado en esta tesis está lejos de la vanguardia, sienta todas las bases para una mejor solución del problema. De tal forma que el diseño de este nos permitió comprender los sistemas del estado actual, de los que se habla en el cuarto capítulo. El cuarto capítulo consta, pues, de una explicación, completa y atendiendo los detalles que a veces se omiten y dificultan el entendimiento, de los artículos en los que fue presentado el sistema KinectFusion, porque sirve como ejemplo para entender trabajos posteriores. También, se presenta un breve resumen de las diferentes áreas del problema que los artículos más nuevos resuelven.

El quinto capítulo es una descripción minuciosa del diseño del sistema que cumple con el objetivo de este trabajo. El capítulo abarca los dos módulos que lo conforman: la construcción del ambiente virtual, y la simulación dentro de ese ambiente. Se habla también de la implementación.

El capítulo sexto describe el experimento realizado para probar el sistema del capítulo quinto. Entre los resultados se encuentran las consideraciones tomadas para optimizar el rendimiento, la comparación contra el CPU realizando el mismo procesamiento, y la medición del error relativo de las mediciones del mapa de profundidad simulado con respecto a las mediciones del mapa de profundidad original capturado por el sensor.

Las conclusiones, capítulo séptimo, incluyen también algunas sugerencias del vasto trabajo a futuro.

Cap´tulo 2

Cómputo en la GPU

En la actualidad, los programadores podemos utilizar las GPU, a través de plataformas como CUDA y OpenCL, como coprocesadores de la CPU, viéndolas como procesadores numéricos paralelos, sin preocuparnos o tener conocimiento de temas de computación gráfica que en otro tiempo eran necesarios para utilizar estos dispositivos con el fin de realizar cómputo paralelo. Sin embargo, conocer un poco de la historia de cómo pasaron de ser procesadores gráficos, exclusivamente, a procesadores de propósito general, permite entender las fortalezas y debilidades de estos sistemas. Permite, sobre todo, entender características del diseño, como su capacidad masiva de procesamiento numérico, el caché limitado en comparación con una CPU, y el compromiso en la transferencia de datos CPU-GPU.

2.1. Breve historia

La industria implacable de los videojuegos, con su necesidad de satisfacer la demanda del mercado de tener escenas interactivas complejas y con creciente similitud a la realidad; la reducción de tamaño de transistores y de los circuitos integrados; las mejoras en las técnicas de producción y diseño de *hardware*; y el trabajo en algoritmos de gráficos por computadora, provocan, en conjunto, un avance de magnitud tal que, de la década de los ochenta del s.XX a la mitad de la segunda década del s.XXI, las unidades de procesamiento gráfico han pasado de ser máquinas enormes a ser tarjetas incrustables en una computadora de escritorio, o circuitos del tamaño de una moneda en una computadora portátil; han dejado de costar decenas de miles de dólares[2] para costar, ahora, solamente cientos, o miles; han aumentado su capacidad de generar cincuenta millones de pixeles por segundo a cien mil millones de pixeles por segundo; y más importante, han dejado de ser *pipelines* fijos para desplegar modelos geométricos como uniones de líneas, para convertirse en arquitecturas altamente programables, capaces de procesar escenas tridimensionales complejas y datos en general.

De inicios de los ochenta hasta finales de los noventa, el *hardware* de gráficos era un *pipeline*[1] con elementos configurables, pero no programables. En esa misma época, las API (*Application Programming Interface*) que permitían configurar los elementos del hardware comenzaron a surgir, persistiendo dos: OpenGL, basada en la API de los equipos de la compañía *Sillicon Graphics*, y liberada en 1992 por esta compañía con la intención de que se convirtiera en una manera normalizada, y multiplataforma, de escribir aplicaciones con gráficos tridimensionales; y Direct3D, parte de la API DirectX para el manejo multimedia de Microsoft. En esos sistemas, un programa enviaba comandos, determinados por el programador a través de la API, a una interfaz entre el CPU y el *hardware* gráfico. Un ejemplo de esta época es la GeForce 256, que apareció en 1999, y fue la primera unidad llamada GPU. Esa GPU era capaz de realizar las transformaciones sobre los vértices y la iluminación utilizando el hardware con las unidades dedicadas para esas tareas; de realizar operaciones comunes como asignar colores, normales, o coordenadas de textura a los vértices de un triángulo; y de asignar, mediante interpolación, el color final a cada pixel de los triángulos que formaban una escena. Por veinte años, aproximadamente, cada generación de *hardware* y las API correspondientes proporcionaban nuevos elementos configurables en el *pipeline*, pero estas novedades no eran suficientes para el ritmo de los desarrolladores y los nuevos algoritmos, por lo que el paso inmediato fue convertir algunas etapas del *pipeline* en procesadores programables.

Los primeros pasos consistieron en hacer programable la etapa de procesamiento de vértices (transformaciones geométricas de los vértices), cumpliendo con las especificaciones de DirectX 8 (NVIDIA GeForce 3, 2001). Las GPU que cumplían con DirectX 9 (2002) extendían su programabilidad a la etapa de *pixel shading* (cálculo del color de los pixeles). En el *pipeline* gráfico, estas dos etapas realizan una cantidad importante de operaciones de punto flotante sobre datos independientes. Esta característica de los datos es una de las diferencias principales entre las ideas de diseño entre una GPU y una CPU. En una GPU necesitamos *hardware* que nos permita procesar en paralelo un millón de triángulos y seis millones de pixeles en un cuadro de una escena, por ejemplo. Estas unidades programables atrajeron la atención de los investigadores que eran conscientes del poder de cómputo que podrían aportar. En aquel tiempo, la única manera de interaccionar con la GPU era a través de Direct3D y OpenGL, por lo que el procedi-miento para utilizar las unidades programables resultaba engorroso[2] y requería conocer alguna de las API.

[1] El *pipeline* grá co es el conjunto de etapas por las que pasa un pol´gono desde su de nición como un conjunto de vértices en tres dimensiones, hasta el cálculo del color de los pixeles que ocupa en un despliegue.

[2] El procedimiento, en esencia, era el siguiente. Las GPU ten´an como propósito calcular, para una posición `(x,y)` en la pantalla, un color de salida con un *pixel shader*. Ese color era producto de entradas como coordenadas de una textura, colores, normales. No es dif´cil darse cuenta de que los colores de entrada, o los datos de la textura, pod´an ser datos numéricos con otro signi cado que no fuera color. O sea, se utilizaba

El deseo de tener una GPU que tuviera un arreglo de procesadores unificados, en contraste con procesadores diferentes para cada una de las etapas del *pipeline*, se vio materializado en 2006, cuando la GeForce 8800 salió al mercado. El *pipeline* gráfico ahora era lógico, pues todas las etapas eran llevadas acabo en los mismos procesadores. La GeForce 8800 fue pensada, desde su diseño, para ser usada para cómputo de propósito general, y fue la primera en ser construida conforme a la arquitectura CUDA de NVIDIA cuyo objetivo es el de resolver las limitaciones de procesadores anteriores que impedían su uso como procesadores de datos en general.

Aunque ya existía *hardware* con arquitectura similar a la de sistemas paralelos, quedaba la limitación de tener que utilizar una API para gráficos. Para superar esto, la industria desarrolló marcos de trabajo (*frameworks*) y algunas API, entre los que destacan OpenCL, propuesto por Apple Inc. y manejada en la actualidad por Khronos Group como una norma abierta y multiplataforma; y CUDA, de la compañía NVIDIA, de la que hablaremos en adelante por la facilidad de aprendizaje, e incremento de uso en la industria y academia.

2.2. CUDA

CUDA (*Compute Uni ed Device Architecture*), presentada en noviembre de 2006, es una plataforma y modelo de programación que permite utilizar las GPU de la compañía *NVIDIA* como auxiliares al procesamiento de la CPU. Esto se logra a través de una extensión del lenguaje C, conocida como CUDA C; a través de algún otro lenguaje como FORTRAN, Python, o de alguna API como DirectCompute u OpenACC; o a través de bibliotecas como CUBLAS, CUFFT, o Thrust.

Las arquitecturas CUDA, que repasaremos en la siguiente sección, ofrecen *hardware* que se asemeja a sistemas paralelos. Una de las características de estas arquitecturas es que escalan según la ley de Moore[3]. CUDA y su modelo de programación, que será estudiado después de las arquitecturas, permiten escribir aplicaciones que escalan, fácilmente, al crecimiento en el número de núcleos de procesamiento.

2.3. Arquitecturas CUDA

Cuando se diseñó la arquitectura CUDA, se buscaba tener un arreglo de procesadores en el que cada unidad aritmético-lógica (*ALU, Arithmetic Logic Unit*) pudiera ser utilizada para realizar cómputo de propósito general. Para lograr esto, las ALU fueron diseñadas para

el *pixel shader* para manipular esos datos haciéndolos pasar por colores y se enganaba a la GPU para que realizara el cómputo deseado.

[3]El número de transistores se duplica cada dos anos.

cumplir con los requerimientos del Instituto de Ingeniería Eléctrica y Electrónica (*IEEE, Institute of Electric and Electronic Engineers*) de operaciones de punto flotante. Se buscaba, además, que las unidades de ejecución en la GPU pudieran hacer, arbitrariamente, lecturas y escrituras en memoria principal y en una memoria de tipo caché (para acelerar operaciones de memoria) conocida como memoria compartida (*shared memory*).

En *hardware*, esto se logra a través de un arreglo aumentable de procesadores multihilo llamados *Streaming Multiprocessors (SM)*. Los SM están diseñados para ejecutar hilos concurrentemente, y están conformados, principalmente, por procesadores (*CUDA cores*), memoria compartida, unidades de funciones especiales, y registros. Los SM han evolucionado, como veremos, con el paso del tiempo en las diferentes arquitecturas.

2.3.1. Tesla

La arquitectura Tesla[4][3] está basada en un arreglo de procesadores conocido como *streaming processor array (SPA)*. En esta arquitectura, los elementos de este arreglo fueron llamados *texture/processor cluster (TPC)* y la GeForce 8800, por ejemplo, contiene ocho de estas unidades (fig. 2.1).

Cada uno de los TPC está formado por dos *Streaming Multiprocessors (SM)*, que se describen en el siguiente párrafo; por un *Geometry Controller*, que se encarga de relacionar el *pipeline* gráfico a los SM; y por un *Streaming Multiprocessor Controller (SMC)*, que se encarga de manejar los dos SM, distribuir la carga, y administrar la unidad de textura.

Los SM también son arreglos (fig. 2.2), y están formados por ocho *streaming-processors (SP)* (ahora conocidos como *CUDA cores*), que tienen el propósito de realizar operaciones aritméticas, de comparación y de conversión; por dos unidades de funciones especiales (*special function unit, SFU*), que realizan operaciones de funciones trascendentes y de interpolación; por una unidad de captura y emisión de instrucciones multihilo (*multithreaded instruction fetch and issue unit, MT Issue*); por un caché de instrucciones; por un caché constante de solo lectura; y por una memoria compartida de 16 KB de lectura y escritura.

Los SM tienen como finalidad ejecutar, concurrentemente, los programas de diferentes hilos. Un hilo —la unidad mínima de ejecución— puede llevar a cabo las operaciones de un vértice, de un pixel, o de un cómputo cualquiera. El *hardware* multihilo de un SM de esta arquitectura puede manejar y ejecutar hasta 768 hilos concurrentemente. Para hacerlo

[4]Nombrada as´ en honor a Nikola Tesla, el f´sico e ingeniero de origen serbio (nacionalizado estadounidense más tarde en su vida) que vivió de 1856 a 1943 y que es reconocido por sus trabajos innovadores en corriente alterna, como el motor de inducción, que fue básico en los primeros disenos del suministro de energ´a electrica.

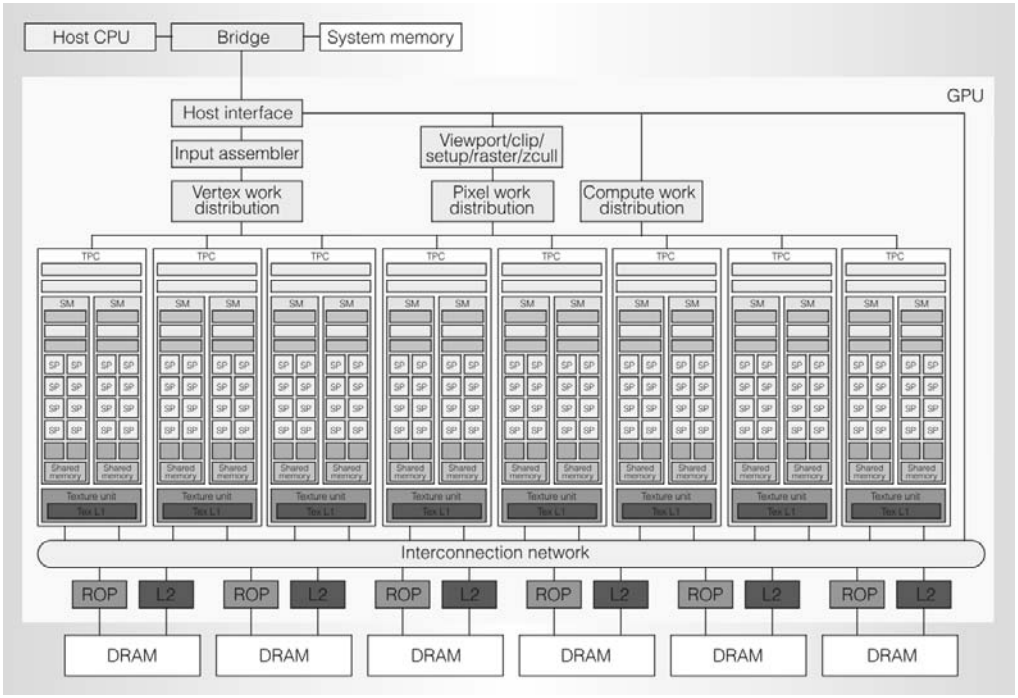


FIGURA 2.1: La arquitectura Tesla presentada en [3] como una arquitectura unificada de cómputo y gráficos.

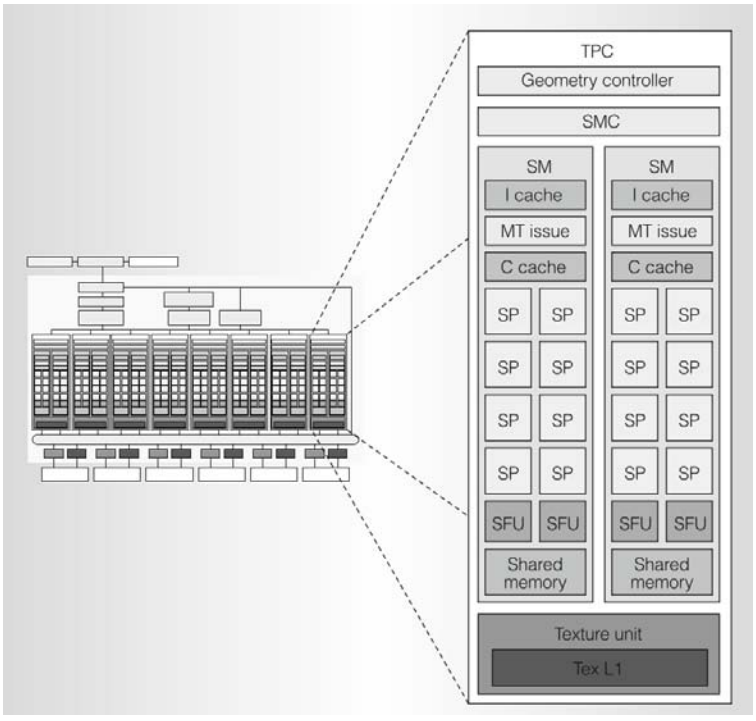


FIGURA 2.2: Los elementos que conforman un TPC y un SM. [3]

de manera eficiente, un SM Tesla utiliza una arquitectura de procesador llamada *single-instruction, multiple-thread (SIMT)*. La unidad SIMT multihilo del SM crea, maneja, planifica y ejecuta hilos de forma paralela en grupos de 32 llamados *warps*. Esta unidad elige un *warp* que esté listo para ser ejecutado y envía a todos los hilos de ese *warp* la siguiente instrucción a ejecutarse[5].

2.3.2. Fermi

Después de la arquitectura Tesla, los diseñadores de *hardware* de *NVIDIA* tomaron las bases del procesador unificado, analizaron las diversas aplicaciones que se realizaron para estos dispositivos, y utilizaron las recomendaciones de los usuarios para mejorar la arquitectura. El resultado de este enfoque fue la arquitectura Fermi[6] presentada en 2010 [4, 5]. Entre las mejoras sustanciales podemos encontrar el soporte para corrección de errores (*Error Correction Code, ECC*), para asegurar la integridad de los datos en aplicaciones sensibles; la mejora en el rendimiento de operaciones de doble precisión, para aplicaciones de cómputo científico; jerarquía de caché, para aumentar la eficiencia en lecturas de memoria; más memoria compartida; y operaciones atómicas en memoria.

En esta arquitectura los SM fueron modificados de manera importante (fig. 2.3). Pasaron de tener 8 *CUDA cores* que utilizaban la norma IEEE 754-1985 a tener 32 que cumplen la norma IEEE 754-2008 y son capaces de realizar una operación de suma y multiplicación con un solo paso de redondeo (en contraste con realizar la multiplicación y redondear, y después realizar la suma y redondear). Los *CUDA cores* también pueden realizar operaciones con enteros de 32 y 64 *bits*, así como operaciones de doble precisión en punto flotante. Se integraron, también, 16 unidades de carga y almacenamiento (*Load/Store)* que permiten el cálculo de direcciones de memoria para 16 hilos por ciclo de reloj. Aumentaron en dos las SFU. El SMC de la arquitectura Tesla fue sustituido por dos planificadores de *warps* que pueden emitir y ejecutar una instrucción de dos *warps* de manera concurrente. Una de las novedades más atractivas de esta arquitectura fue la inclusión de 64 KB de una memoria configurable como memoria compartida o como caché de tipo L1, que hace innecesario el uso de memoria compartida para la mayoría de los *kernels*.

Los SM se agrupan con otros elementos para formar la arquitectura Fermi (fig. 2.4). Los grupos de SM (que pasaron de llamarse TPC a *Graphics Processing Clusters* o GPC)

[5] Algunos hilos pueden permanecer inactivos por bifurcaciones en el código, por ejemplo, cuando hay código condicional. El rendimiento máximo suele alcanzarse cuando los hilos de un mismo *warp* ejecutan exactamente la misma instrucción, aunque sobre diferentes datos

[6] Por Enrico Fermi, un f´sico italiano, premio Nobel de esta área en 1938, que vivió de 1901 a 1954 y que contribuyó en la mecánica cuántica, la f´sica nuclear y de part´culas, y la mecánica estad´stica. Participó como l´der de proyecto en el desarrollo del primer reactor nuclear.

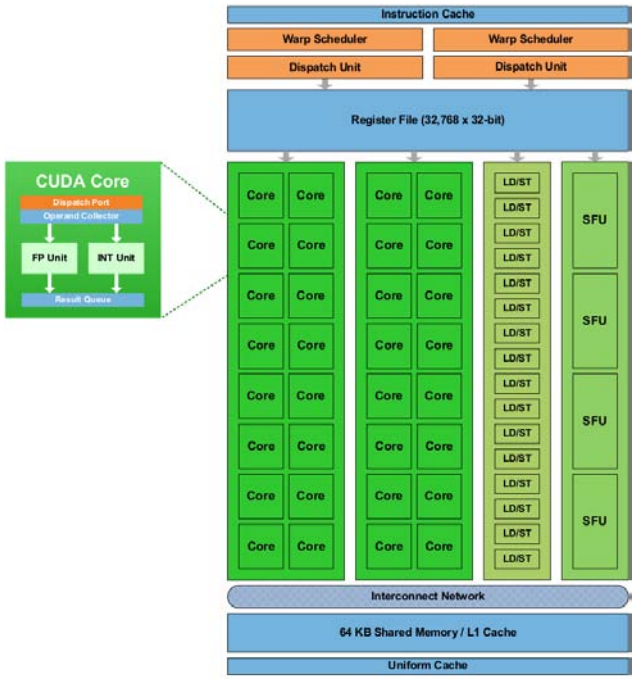


FIGURA 2.3: La estructura de un SM de la arquitectura Fermi. [5]

rodean un caché de nivel L2 de lectura y escritura utilizable por los SM, los *raster engines* y los *polymorph engines*. La arquitectura cuenta con seis particiones de memoria de 64 *bits*, que permiten la lectura de 384 *bits* simultáneamente de una memoria GDDR5 de hasta 6 GB. Los hilos, en esta arquitectura, son planificados en dos niveles: en el nivel superior la distribución de trabajo se lleva a cabo por el módulo llamado *GigaThread*, que planifica y distribuye bloques de hilos hacia diferentes SM; y en un nivel inferior, los planificadores de *warps* (*warp scheduler*), dividen los bloques de hilos y los organizan en grupos de 32 hilos para su ejecución dentro de un SM.

2.3.3. Kepler

La arquitectura Kepler[7], presentada en 2012 [6], toma, en gran medida, características de su antecesora. Sin embargo, incorpora cambios que potencian la GPU como herramienta auxiliar en el procesamiento de datos y cálculo de operaciones en punto flotante.

Los nuevos SM fueron llamados *Next Generation Streaming Multiprocessors (SMX)*. La principal modificación fue el aumento de *CUDA cores*, de 32 a 192. Las unidades *load/store* se duplicaron, pasando de 16 a 32. Cada SMX tiene 4 planificadores de *warps* y cada uno

[7]Por Johannes Kepler, el matemático, astrónomo y profesor alemán que vivió de 1571 a 1630. Sus leyes del movimiento planetario, que mejoraban la teor´a heliocéntrica de Copérnico, servir´an para el desarrollo de la ley de la gravitación universal de Isaac Newton más tarde.

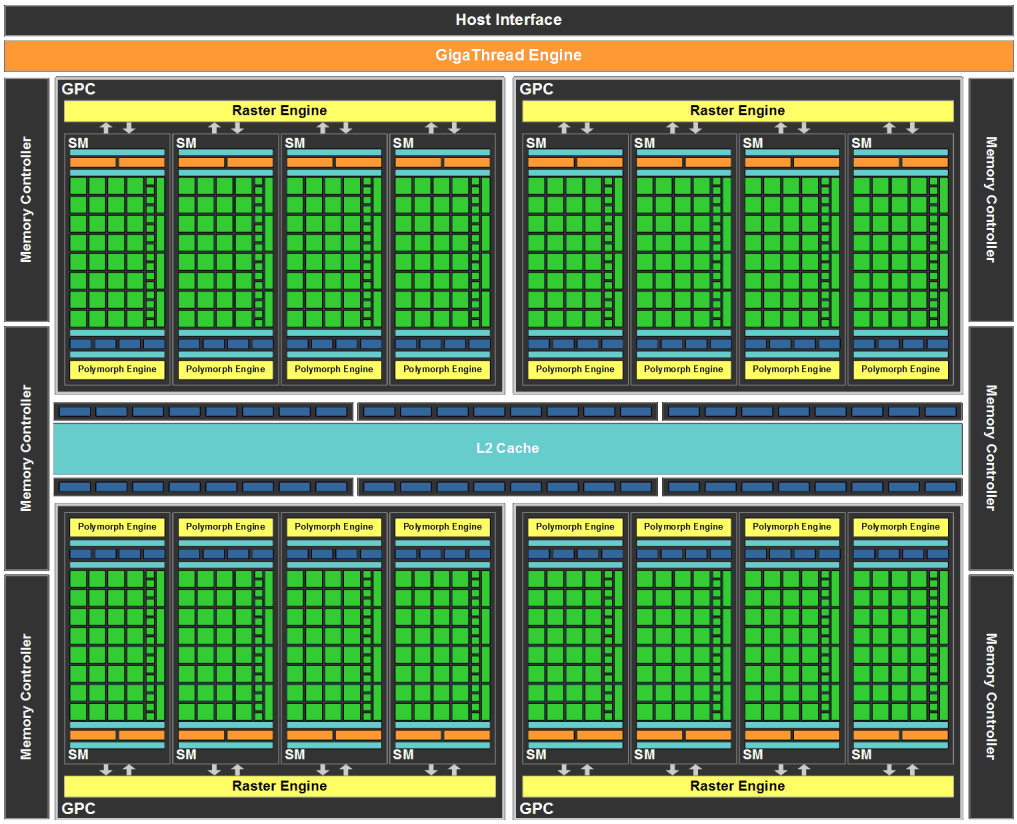


FIGURA 2.4: Esquema de la arquitectura Fermi. [4]

de ellos tiene dos unidades de despacho de instrucciones, permitiendo que cada planificador ejecute, concurrentemente, una instrucción de dos *warps* diferentes. Las SFU se cuatriplicaron, llegando a 256. El caché L1 puede ser configurado para ser de 16, 32 o 48 KB, dejando el resto para memoria compartida. Finalmente, se duplicó el número de registros, llegando a 65 536.

En una revisión en 2013[7] de la arquitectura Kepler, identificable con el código GK110[8], y enfocada, principalmente, hacia las GPU para el cálculo de alto desempeño, se agregaron 64 unidades de doble precisión (*DP Units*). En esta arquitectura, cada hilo puede utilizar 255 registros para almacenar datos. Desapareció el concepto de caché de textura y se sustituyó por un caché de solo lectura de 48 KB que agiliza las lecturas a memoria global o que funciona como caché de textura cuando se hacen lecturas de este tipo. El *polymorph engine*, que se encargaba de realizar operaciones gráficas como lectura de vértices, teselado,

[8]Hasta este momento no hab´amos revisado estos códigos, pero permiten identi car las arquitecturas y sus revisiones. La letra que sucede a la ge mayúscula, es la primera letra de la arquitectura en cuestión, y el número indica la versión dentro de esa arquitectura. Por ejemplo, la descrita en la sección pasada correspond´a al código GF100, y la primera de esta sección al código GK104.

asociación de atributos, y cálculo del punto de vista, fue removido con la idea de hacer la arquitectura enteramente programable.

En esta arquitectura se introdujo la capacidad de generar nueva carga de trabajo desde un programa ejecutable en GPU, administrar recursos, y manejar los resultados sin la intervención de la CPU. Esta característica, llamada *Dynamic Parallelism*, puede resultar de utilidad, por ejemplo, en una simulación numérica en la que no se requiere el mismo detalle en toda la malla, sino que en zonas de interés se necesitan más cálculos.

2.3.4. Maxwell

Esta arquitectura, presentada en 2014 con el nombre Maxwell[9][8], también es un arreglo de GPC, cada uno con cuatro SM, alrededor de un caché L2 que incrementó su tamaño de 512 KB en GK104 a 2048 KB en GM204, y con cuatro controladores de memoria.

El SM en esta arquitectura se llama *Maxwell Streaming Multiprocessor* (SMM) y tuvo algunas modificaciones. Dispone de 128 *CUDA cores*, dispuestos en arreglos de 32 con sus propios planificadores de *warps* y caché de instrucciones (fig. 2.5). Esta nueva disposición es coherente con el tamaño de los *warps* y acelera la transferencia y planificación de ellos. La jerarquía de memoria también cambió, ahora el caché L1 se comparte con el caché de textura, y la memoria compartida se incrementó de 64 KB a 96 KB.

La tabla 2.1 resume las características importantes de las arquitecturas y nos muestra la clara tendencia en el aumento de las capacidades de estos procesadores y de ser, cada vez más, herramientas benéficas para la aceleración en el procesamiento de datos.

2.4. Modelo de programación de CUDA

El modelo de programación de CUDA es la abstracción que nos permite utilizar el *hardware* de la GPU para realizar operaciones de cómputo auxiliares al procesamiento en CPU. Está basado, principalmente, en el modelo de un programa y múltiples datos (SPMD, *Single Program, Multiple Data*). Al programa que se ejecutará sobre múltiples datos se le conoce como *kernel*.

[9] En honor a James Clerk Maxwell, el cient´ co y matemático escocés que vivió de 1831 a 1879 que unicó los modelos de electricidad y magnetismo con su elegante y conocido conjunto de ecuaciones diferenciales parciales.

[10] En las caracter´sticas de NVIDIA puede encontrarse el doble de las cantidades aqu´ reportadas, pero se aprovechan del hecho de que la memoria es *Double Data Rate*.

FIGURA 2.5: Comparación del SM en las arquitecturas Kepler (izquierda) y Maxwell (derecha). [6, 8]

	Tesla GT200	**Fermi** GF110	**Kepler** GK104	**Maxwell** GM204
Total de SM	30	16	8	16
***CUDA cores* por SM**	8	32	192	128
CUDA cores	240	512	1536	2048
Reloj	648 MHz	772 MHz	1006 MHz	1126 MHz
GFLOP	1063	1581	3090	4612
Unidades de textura	80	64	128	128
Reloj de la memoria [10]	2484 MHz	4000 MHz	6000 MHz	7000 MHz
Ancho de banda de la memoria	159 GB/s	192.4 GB/s	192 GB/s	224 GB/s
Consumo de energ´a	184 W	244 W	195 W	165 W
Transistores	1.4×10^9	3.0×10^9	3.54×10^9	5.2×10^9

TABLA 2.1: Comparación de las características de las diferentes arquitecturas.

2.4.1. Funciones *kernel*

En CUDA C[11], un *kernel* es una función `void` de C, calificada con la palabra reservada `__global__`, que especifica el código que ejecutará cada uno de los hilos sobre los datos (list. 2.1). Los *kernels* se ejecutan en un **dispositivo** (*device*, una GPU instalada en el equipo de trabajo) y se invocan desde un programa en el **equipo an trión** (*host*, el equipo desde donde se ejecuta el programa que utiliza el coprocesador).

```
1 __global__ void miKernel(...){
2     ...
3 }
```

Listado 2.1: Ejemplo de la definición de un *kernel*.

Al lanzar[12] un *kernel* desde la aplicación del equipo anfitrión, se elige una configuración de hilos. Estos hilos son planificados y procesados en la GPU, llevando a cabo, como ya dijimos, la tarea escrita en el *kernel*.

En la siguiente sección veremos que los hilos (*threads*) son agrupados en bloques (*blocks*), y los bloques, a su vez, en una malla (*grid*) para la ejecución.

2.4.2. Jerarqu´a de hilos

Al llamar un *kernel* dentro de código que se ejecuta en el equipo anfitrión, se especifica, después del nombre, una configuración de lanzamiento entre <<< y >>>. Dentro de estos símbolos se escriben las dimensiones de la malla y de los bloques (list. 2.2).

Una malla contiene bloques, y puede ser uni, bi o tridimiensional. Los bloques, que también pueden ser de una, dos, o tres dimensiones, agrupan hilos (fig. 2.6). El número de hilos que pueden lanzarse para ejecutar un *kernel* puede consultarse en la tabla 2.2.

```
1 int main(...){
2     ...

4     dim3 gridDim(..., ..., ...);
5     dim3 blockDim(..., ..., ...);

7     miKernel <<< gridDim,blockDim >>>(...);

9 }
```

Listado 2.2: Ejemplo del lanzamiento de un *kernel*.

[11] Si el lector está familiarizado con C, CUDA C es la manera más fácil de aprender a programar utilizando la GPU. Recuerde que hay otras interfaces y se pueden utilizar otros lenguajes de programación.

[12] Quizá sea un abuso del lenguaje, pero es común utilizar este verbo para referirse al llamado de un *kernel*.

Dado que todos los hilos de la malla ejecutan un mismo *kernel*, se necesita de un identificador único para que los hilos sean distinguibles unos de otros y puedan efectuar operaciones sobre diferentes porciones de los datos. El modelo provee identificadores para los hilos, `threadIdx`, y para los bloques, `blockIdx`, de manera que un hilo puede conocer su posición —en cada dimensión— dentro del bloque al que pertenece y la posición del bloque dentro de la malla. Además, un hilo puede conocer las dimensiones del bloque (número de hilos), mediante la variable `blockDim`, y de la malla (número de bloques), a través de la variable `gridDim`.

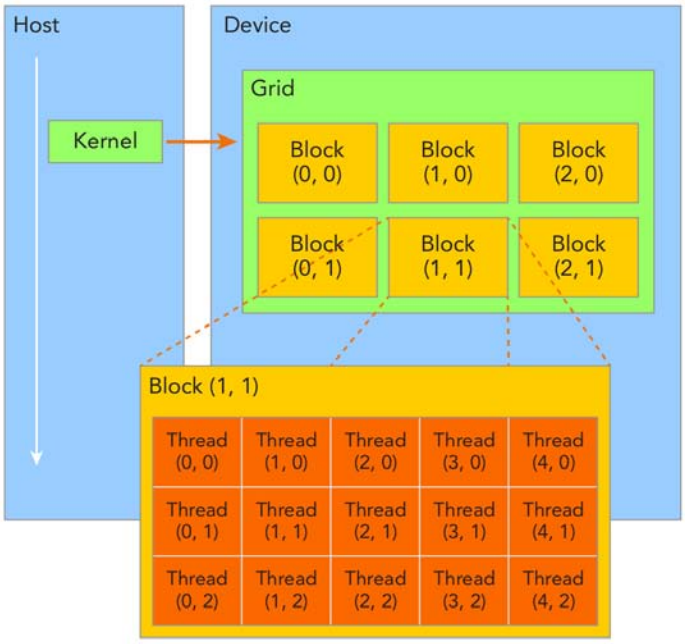


FIGURA 2.6: Organización de los hilos. Una malla contiene bloques, y los bloques contienen hilos. En el ejemplo, la malla tiene tres bloques en la dirección x, y dos en la dirección y, y los bloques tienen cinco hilos en la dirección x, y tres en la dirección y.[9]

La cooperación entre hilos es posible, solamente, si pertenecen al mismo bloque. La función `__syncthreads()` proporciona esta cooperación a través de una sincronización de barrera, o sea, un punto de la ejecución al que todos los hilos de un mismo bloque deben llegar antes de continuar. La cooperación entre hilos suele ser más efectiva utilizando la memoria compartida (*shared memory*), de la que se hablará en la siguiente sección. La decisión de diseño de CUDA de no permitir sincronización entre bloques provoca que la ejecución de los bloques pueda ocurrir en cualquier orden y que estos sean asignados a cualquier SM, logrando con esto, además de la máxima utilización de recursos, que las

aplicaciones se ejecuten sin dificultad en dispositivos con diferente número de SM (v. Fig. 2.7).

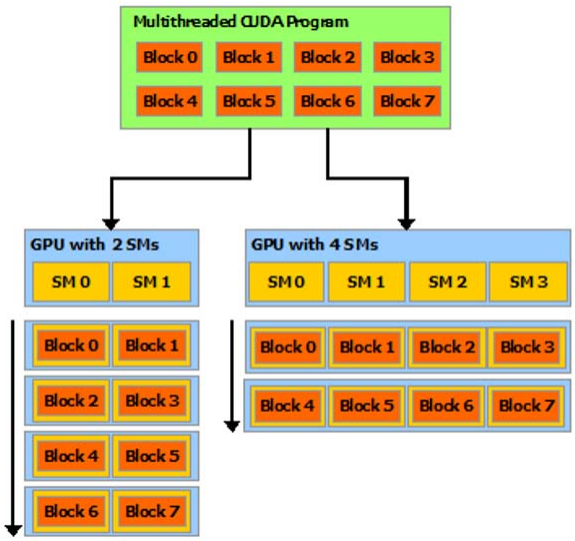


FIGURA 2.7: Los bloques de hilos son asignados a diferentes SM. [1]

2.4.3. Jerarqu´a de memoria

Un *kernel*, como el que describimos en la sección anterior, realiza operaciones sobre datos que lee de, y escribe en, algún lado. En el modelo de programación de CUDA, los hilos acceden a diferentes espacios de memoria (fig. 2.8) con características diversas de las que hablaremos ahora.

La memoria global (*global memory*), que es memoria del dispositivo, es la de mayor tamaño —es común encontrar de dos o cuatro GB— y la que funciona como lectura de datos y escritura de resultados, es, también, la interfaz entre el CPU y el GPU y entre diferentes *kernels*. Se reserva, maneja y libera a través de llamadas de funciones desde el equipo anfitrión: `cudaMalloc`, `cudaMemCpy`, `cudaFree`. La memoria global puede ser accedida por cualquier hilo en cualquier parte de un *kernel*. No obstante, su alta latencia puede provocar que los SM pasen más tiempo esperando una lectura o escritura, que haciendo cálculos.

La memoria constante (*constant memory*) está optimizada para enviar datos de solo lectura a múltiples hilos. Reside en el dispositivo, como la memoria global, pero utiliza instrucciones diferentes que permiten el acceso a este caché constante de forma más rápida que una lectura a memoria global. El programador tiene a su disposición 64 KB para variables constantes que se declaran afuera de cualquier función con la palabra reservada `__constant__`, y que pueden ser modificadas dentro de un *kernel* (siempre y cuando no estén

siendo utilizadas por otro) o pueden guardar datos que se copien desde el equipo anfitrión con la función `cudaMemcpyToSymbol`.

La memoria de textura (*texture memory*) permite almacenar en memoria de dispositivo un arreglo de bytes llamado arreglo CUDA (CUDA *array*) con un patrón que permite el acceso eficiente a información de una, dos, o tres dimensiones (un arreglo, una imagen, y un volumen, por ejemplo).

La memoria compartida (*shared memory*) permite el intercambio de datos entre hilos de un mismo bloque. Es hasta diez veces más rápida de acceder que la memoria global, pues cada SM cuenta con este tipo de memoria. Es de 48 KB por bloque a lo más, y es manejable manualmente por el programador (como un caché manual). Las variables de este tipo se especifican con la palabra reservada `__shared__` y pueden ser leídas y modificadas por hilos que pertenecen a un mismo bloque.

La memoria local (*local memory*) contiene la pila para cada hilo de un *kernel*. Guarda datos que no pudieron ser guardados en registros y es administrada por el sistema.

Los registros son la memoria más rápida de la que dispone un hilo y son asignados a estos al momento de lanzar un *kernel*. Cada SM contiene miles (65 536, para Kepler, por ejemplo) de registros de 32 bits y pueden contener información de punto flotante o de enteros.

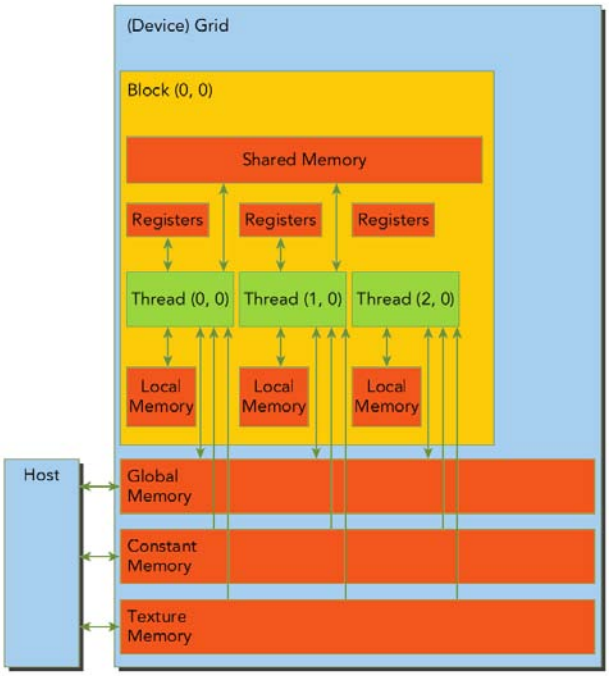


Figura 2.8: Los diferentes espacios de memoria disponibles en CUDA. Las flechas representan desde qué contexto pueden ser accedidos. [9]

2.4.4. Programación heterogénea

El modelo de programación de CUDA asume que los hilos de un *kernel* se ejecutan en un dispositivo que opera como coprocesador al CPU del equipo anfitrión que está ejecutando el programa en CUDA C. También asume que el dispositivo y el equipo anfitrión tienen su propios espacios de memoria separados, conocidos como memoria en equipo anfitrión (*host memory*) y memoria en dispositivo (*device memory*). El programa del equipo anfitrión maneja las memorias global, constante y de textura que están en GPU y son visibles para los *kernels*, incluyendo la reservación, transferencia, y liberación de esos espacios de memoria.

Un programa en CUDA C tendrá, generalmente, la estructura que se muestra con un remedo de código en el listado 2.3, y que se describe a continuación:

Se declaran y definen funciones *kernel* y funciones ejecutables en GPU y llamables desde un *kernel* que se definen con la palabra reservada `__device__`.

En el programa principal, se reserva memoria en el equipo anfitrión o se obtienen los datos que serán la entrada a un *kernel* y serán procesados en la GPU.

Se reserva memoria en el dispositivo, del tamaño y tipo de los datos que serán procesados en la GPU y de los resultados que se obtendrán de este procesamiento.

Se copian los datos de entrada del equipo anfitrión al dispositivo.

Se ejecuta uno o varios *kernels* sobre los datos en la GPU, eligiendo las dimensiones de los bloques y de la malla con la sintaxis descrita anteriormente.

Se copian los resultados del procesamiento en el dispositivo hacia el equipo anfitrión para continuar su tratamiento.

Se liberan los recursos que ya no serán utilizados en el dispositivo.

Continúa el procesamiento y el flujo del programa en CPU.

```
1  __device__ T funA(...){
2      ...
3  }

5  __global__ void kernelA(T*, ... ){
6      ...
7      T a=funA(...);
8      ...
9  }

11 __global__ void kernelB(...){
12     ...
13 }

15 int main(...){
16     ...
17     T* dataDevice;
18     cudaMalloc(&dataD,sizeData);
```

```
19    ...

21    cudaMemcpy(dataDevice,src,sizeToCopy,cudaMemcpyHostToDevice);

23    ...
24    dim3 gridDim(gx,gy,gz);
25    dim3 blockDim(bx,by,bz);

27    kernelA <<< gridDim,blockDim >>>(dataDevice, ... );
28    ...
29    kernelB <<< ..., ... >> ( ... );
30    ...

32    cudaMemcpy(res,dataDevice,sizeToCopy,cudaMemcpyDeviceToHost);
33    ...

35    cudaFree(dataD);
36    ...
37 }
```

LISTADO 2.3: Estructura general de un programa en CUDA C.

2.4.5. *Compute capability*

La capacidad computacional (*compute capability*) de un dispositivo está indicada por un número **x.y** donde **x** representa la versión mayor y **y** la versión menor.

Este número identifica las capacidades del *hardware* de una GPU y las aplicaciones pueden utilizarlo para saber qué funciones están disponibles.

Los dispositivos que tienen un mismo número **x** de versión, comparten la misma arquitectura. Los dispositivos con arquitectura Maxwell son identificables con el número 5, aquellos con Kepler con el 3, Fermi con el 2 y Tesla con el 1. El número **y** indica una mejora en la arquitectura y nuevas características.

Conocer la capacidad computacional del dispositivo con el que se cuenta, permite determinar fácilmente los recursos con los que cuenta el dispositivo (tabla 2.2).

2.5. Optimización de rendimiento

El propósito de utilizar una GPU como auxiliar de la CPU, o de paralelizar una aplicación, es reducir los tiempos de ejecución de los programas. Lograr esto no es tarea sencilla, y la inversión de tiempo o recursos económicos requeridos hace necesario un análisis minucioso del problema para elegir una estrategia adecuada que optimice una aplicación de *software*.

Consideramos que entre los puntos clave que hay que reconocer están:

	Compute Capability							
Especi caciones Técnicas	1.1	1.2	1.3	2.x	3	3.5	5	5.2
Número máximo de dimensiones para una malla de bloques	2			3				
Número máximo de bloques en la dimensión x	65535				$2^{31}-1$			
Número máximo de bloques en la dimensión y o z	65535							
Número máximo de dimensiones para un bloque de hilos	3							
Número máximo de hilos en la dirección x o y	512			1024				
Número máximo de hilos en la dirección z	64							
Número máximo de hilos por bloque	512			1024				
Tamaño de un *warp*	32							
Número máximo de bloques residentes por SM	8				16	32		
Número máximo de *warps* residentes por SM	24	32		48	64			
Número máximo de hilos residentes por SM	768	1024		1536	1024			
Número máximo de registros de 32 *bits* por SM	8 K	16K		32K	64 K			
Número máximo de registros de 32 *bits* por hilo	128			63	255			
Cantidad máxima de memoria compartida por SM	16 KB			48 KB		64 KB	96 KB	
Cantidad máxima de memoria compartida por bloque	16 KB			48 KB				
Cantidad de memoria local por hilo	16 KB			512 KB				
Tamaño de la memoria constante	64 KB							

Tabla 2.2: Características según la capacidad computacional.

- Entender el problema y poder separarlo en las partes secuenciales y paralelas.
- Entender el impacto de las operaciones a memoria y tomar acciones para maximizar el desempeño.
- Conocer y comprender las capacidades de cómputo de la GPU.
- Maximizar la ocupación de la GPU.
- E identificar los cuellos de botella utilizando herramientas.

En las siguientes secciones discutimos, brevemente, cada uno de estos puntos.

2.5.1. Separación del problema en partes secuenciales y paralelas

Una de las primeras consideraciones antes de paralelizar un programa es evaluar si la inversión de tiempo que requiere vale la pena. La respuesta es dependiente de la aplicación y del tiempo de ejecución que se considere aceptable para esa aplicación. Si, por ejemplo, la aplicación tarda un tiempo T en ejecutarse, y se desea un tiempo de ejecución $\frac{T}{2}$, conviene más buscar optimizar el programa actual en CPU (para obtener ese factor de aceleración de 2) que invertir tiempo en hacer un sistema CPU y GPU.

Cuando se buscan aceleraciones mayores, es importante separar las partes que se realizarán de forma secuencial y paralela. El modelo de programación de CUDA permite acelerar aplicaciones que presentan paralelismo de datos, o sea, que un grupo de miles o millones hilos puedan trabajar sobre diferentes porciones con poca (o nula) dependencia entre los resultados que produce cada hilo[13]. Al adaptar un problema a este modelo de programación es conveniente pensarlo en términos del conjunto de datos de salida, escoger un elemento y preguntarse si es posible representarlo como una transformación (aplicar fórmulas, operaciones, o procedimientos) sobre elementos de los datos de entrada. Cuando tal transformación es posible, expresar el problema para su solución en GPU es más o menos fácil[14].

Las aplicaciones suelen tener una parte secuencial que limita la aceleración[15] máxima de una aplicación, como indica la ley de Amdahl:

$$A(N) = \frac{1}{(1-P) + \frac{P}{N}} \tag{2.1}$$

Donde A es la aceleración, N el número de hilos o procesadores, y $P \in [0, 1]$ la proporción paralelizable del programa. Observamos que en el límite, cuando N tiende a infinito, la aceleración tiende a $\frac{1}{(1-P)}$.

Identificar los componentes del problema que sacan provecho de la GPU y aquellos que se benefician más de la estructura de memoria de la CPU es un paso básico en el diseño y la optimización de un sistema de cómputo heterogéneo.

2.5.2. Consideraciones de las operaciones de memoria

Cuando desarrollamos una aplicación en GPU nos interesan dos conceptos relacionados con la memoria: el ancho de banda, que es la cantidad de datos transferibles de un origen a un destino por unidad de tiempo, y la latencia, que es el tiempo que toma efectuar una operación en memoria.

En sus orígenes, las GPU necesitaban transferir información de vértices, colores y textura a altas velocidades, por lo que su diseño se enfocó en tener alto ancho de banda a una latencia considerable. Esta latencia puede mitigarse y hasta ocultarse mediante el mecanismo a continuación descrito. Cuando un hilo requiere información que no está lista, se ejecutan los hilos de otro *warp* y, mientras, el *hardware* emite una petición de lectura o

[13] El ejemplo más claro y sencillo de paralelismo de datos es la suma vectorial. Imagine el lector dos arreglos `a` y `b` de dimensión N, el resultado `c[i]=a[i]+b[i]` puede ser calculado, independientemente, por un hilo para cada `i`.

[14] En la suma vectorial es muy fácil, un elemento del conjunto de datos de salida `c[i]` puede obtenerse de la transformación (la suma) de un elemento del arreglo `a` y uno del arreglo `b`.

[15] La razón del tiempo de ejecución de un programa secuencial al tiempo de ejecución de ese programa paralelizado, *speedup* en inglés.

escritura que se combina con otras pendientes del mismo *warp* (siempre y cuando los accesos a memoria sean adyacentes) para obtener la información requerida. Entonces, mientras un SM tenga *warps* cuya información esté disponible y tenga instrucciones por procesar, la latencia quedará oculta. Más adelante discutiremos una manera de mantener los SM ocupados.

En las arquitecturas Fermi y Kepler las acciones sobre la memoria global son a través de líneas de caché L1 de 32 o 128 *bytes*. La manera de aprovechar el ancho de banda es haciendo que cada lectura sea efectiva. Por ejemplo, cuando cada hilo de un *warp* lee una variable de 4 *bytes* (un entero de 32 *bits* o un flotante de precisión simple) de localidades contiguas y alineadas con múltiplos de 32 *bytes*. El acceso ideal a memoria (fig. 2.9) está alineado con múltiplos de 32 *bytes* y es a un grupo de datos contiguos. En inglés este acceso está *aligned and coalesced*. Un acceso puede estar alineado pero no ser a localidades contiguas, o no estar alineado y ser a localidades contiguas, y en esos dos casos se usan más líneas de caché y más accesos a memoria global.

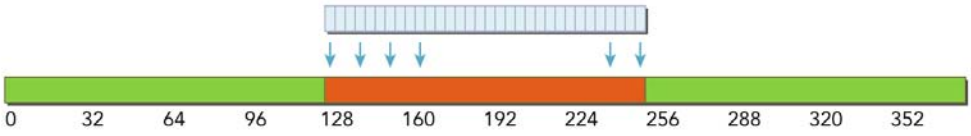


FIGURA 2.9: El acceso ideal a la memoria.[9]

Naturalmente, mientras menos operaciones de memoria se hagan, mayor rendimiento tendrá el programa. En este sentido, en [10] recomiendan una razón de diez instrucciones u operaciones aritméticas por cada lectura a memoria global.

Utilizar la memoria compartida puede acelerar el tiempo de lectura hasta diez veces respecto a la memoria global, pero es limitada en tamaño, pues solamente se dispone de 48 KB. El caché de nivel L1 produce tiempos de lectura similares a la memoria compartida y tiene la ventaja de ser manejado automáticamente por el *hardware* aunque su tamaño también es limitado (configurable, de 16 a 48 KB).

2.5.3. Capacidades computacionales de la GPU

Como ya hemos visto, los bloques se envían a diferentes SM para ser procesados, y una vez en estos, se dividen en grupos de 32 hilos llamados *warps*. Los hilos de un *warp* son ejecutados simultáneamente, por lo que, idealmente, deberían ejecutar la misma instrucción. De otra manera, ese *warp* será enviado a ejecución por el planificador hasta que todos los hilos dentro del *warp* hayan terminado su ejecución, provocando una serialización en las

ejecuciones. En la práctica esto no es tan fácil de hacer porque los hilos de un *warp* pueden tomar caminos diferentes debido a bifurcaciones en el código con `if-else`. Sin embargo, y según sea posible, la lógica del programa debe planearse para tener *warps* que ejecuten la misma instrucción en un momento dado y con esto obtener el mayor rendimiento.

Utilizar las funciones programadas y las que utilizan las unidades de funciones especiales SFU es una forma de acelerar el desempeño. Las funciones como `sin()` o `log()` se ejecutan en GPU como una serie de multiplicaciones y sumas, y su error suele ser, comparado con un redondeo adecuado, de 2 a 3, en general, y no más de 11 ULP[16]. Para utilizar las funciones programadas en las SFU, puede utilizarse la bandera `-use_fast_math` al compilar, o llamarlas desde código ejecutable en dispositivo agregando dos guiones bajos antes del nombre de la función, por ejemplo, `__sin()` o `__log()`. En [11] puede encontrarse información completa sobre estas funciones y el error de aproximación.

Si la aplicación no requiere de números en punto flotante de doble precisión, es preferible utilizar precisión simple. Las operaciones de doble precisión pueden tomar el doble de tiempo (o hasta ocho veces más, en las tarjetas GeForce) de ejecución que las de precisión simple. Claramente, el compromiso entre velocidad y precisión puede compararse con los cálculos para un subconjunto de los datos y determinar si los resultados de precisión simple son aceptables.

Un *kernel* es, a fin de cuentas, una función que puede ser optimizada con métodos como desenrollo de ciclos (*loop unrolling*), unión de ciclos (*loop fusion*), etc[17]. También puede utilizarse el conocimiento del programa y del problema, y la experiencia del programador, para reducir el número de instrucciones y mejorar el rendimiento.

2.5.4. Aumento de la ocupación de los SM de una GPU

Como vimos en las consideraciones de memoria, la latencia puede ocultarse si hay suficientes *warps* ejecutando instrucciones que no requieran accesos a memoria. Para que esto suceda, los SM deberán tener, obviamente, suficientes *warps* para ejecutar. El concepto de ocupación de un SM define la razón de *warps* activos al número máximo de *warps* que puede procesar un SM.

NVIDIA proporciona una hoja de cálculo llamada *CUDA Occupancy Calculator*[12] que permite explorar la ocupación con diferentes configuraciones de lanzamiento para las dimensiones de un bloque. Solamente necesita el número de hilos por bloque, el uso de

[16] *Units at the last place*, es una medida de error entre representaciones de punto otante que compara el d´gito menos signi cativo de dos coe cientes. Por ejemplo, entre las representaciones de π con tres d´gitos signi cativos 0.314×10^1 y 0.315×10^1 existe 1 ULP.

[17] Para información de estas técnicas en el contexto de la programación con GPU se sugiere revisar el cap´tulo 3 de [9].

memoria compartida y el número de registros por hilos (estos dos últimos números pueden obtenerse utilizando la bandera `--ptxas-options=-v` al compilar con `nvcc`), y el resultado sugerido puede ayudar a mejorar el rendimiento.

2.5.5. Identi cación de cuellos de botella mediante herramientas

Una herramienta útil en la identificación de cuellos de botella en una aplicación y problemas de rendimiento en un *kernel* es el perfilador visual de NVIDIA (*NVIDIA Visual pro ler* [13]). Permite analizar, de manera gráfica (fig. 2.10), el tiempo de ejecución de las diferentes partes de un programa, como las transferencias de datos de CPU a GPU, transferencias internas de la GPU, tiempo de ejecución de los *kernels*, etc.

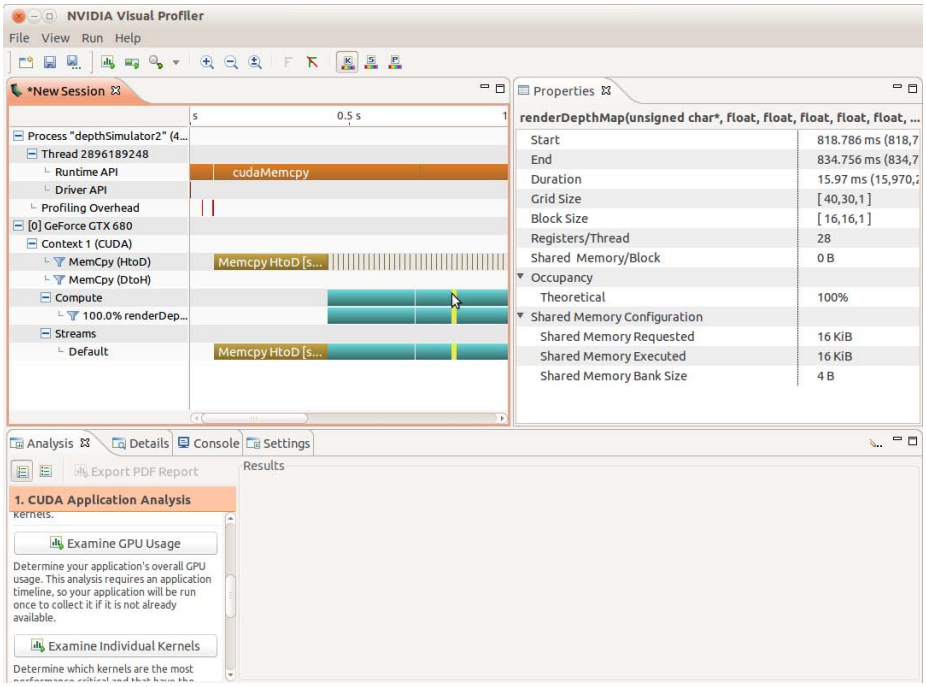


FIGURA 2.10: Perfilador visual de NVIDIA.

También proporciona información más detallada, como la ocupación de los SM, las instrucciones emitidas y ejecutadas por ciclo de reloj en los diferentes SM, la divergencia de instrucciones, la variación de los *warps* activos a lo largo de la ejecución del programa, información de los datos y uso del caché, etc. Además, permite la depuración de aplicaciones.

Con esta herramienta se identifican cuellos de botella como la transferencia a través de PCI-E, que puede minimizarse reduciendo las operaciones de copiado entre CPU y GPU, o haciéndolas de forma concurrente mientras se ejecuta un *kernel*; las lecturas a memoria global, que pueden mitigarse tomando en cuenta las consideraciones de la sección 2.5.2; y

las operaciones de cómputo, que pueden optimizarse según lo discutido en la sección 2.5.3 o siguiendo las recomendaciones que hace el perfilador.

2.6. Ejemplo

A continuación presentamos un ejemplo, ligeramente más complejo que un *hola mundo* tradicional, que tiene por propósito esclarecer algunos de los conceptos del modelo de programación e introducir técnicas útiles y usadas comúnmente al programar en GPU.

El ejemplo consiste en obtener el mínimo de una función evaluada en un conjunto de valores de entrada y también el valor que lo produce. El arreglo de entrada se procesará primero en la GPU y la reducción final se llevará a cabo en la CPU. El *kernel* en GPU se encarga, primero, de evaluar para cada elemento del arreglo de entrada una función $f(x)$, después, cada bloque de hilos CUDA encuentra el mínimo dentro de este bloque y, finalmente, el hilo maestro de cada bloque escribe el resultado a memoria global. La función en CPU se encarga de reducir los resultados producidos por cada bloque. El listado completo se encuentra en el apéndice (list. A.1) y en esta sección analizaremos algunos fragmentos importantes para el total entendimiento del ejemplo.

Al inicio de la función `main` encontramos la definición de los datos de entrada: un arreglo de números en punto flotante de precisión simple (l. `81`) con `N` elementos en el intervalo $[-2.0, 2.0)$ y con `extra` elementos (`nan`) añadidos para lograr que el tamaño del arreglo sea múltiplo de `TPB`, el número de hilos por bloque definido en la línea `4`. La constante `numberOfBlocks` representa el número de bloques de `TPB` hilos que se requerirían para procesar el arreglo.

```
75 int main(void) {

77   float *dataHost = NULL;
78   const int N=1024*90+6;
79   const int extra=(N%TPB!=0)?TPB-N%TPB:0;
80   const int numberOfBlocks=N/TPB+(N%TPB != 0);
81   dataHost=array(-2.0f, 2.0f, N, extra);
```

Ahora nos interesa, siguiendo el flujo del listado 2.3, reservar memoria en la GPU para los datos de entrada (`dataDevice`) y los arreglos de salida que contendrán el valor mínimo que encontró cada bloque de `TPB` hilos (`minPerBlockDevice`) y los índices, en el arreglo original, de las entradas que producen los valores mínimos por bloque (`minPerBlockIdxDevice`). Después de cada reserva de memoria verificamos si hubo algún error en la operación.

```
83   float *dataDevice;
84   float *minPerBlockDevice;
85   int *minPerBlockIdxDevice;
```

```
86   cudaError_t e=cudaSuccess;
87   e=cudaMalloc(&dataDevice,sizeof(float)*(N+extra));
88     if (cudaCheckError(e)) exit(EXIT_FAILURE);
89   e=cudaMalloc(&minPerBlockDevice,sizeof(float)*(numberOfBlocks));
90     if (cudaCheckError(e)) exit(EXIT_FAILURE);
91   e=cudaMalloc(&minPerBlockIdxDevice,sizeof(int)*(numberOfBlocks));
92     if (cudaCheckError(e)) exit(EXIT_FAILURE);
```

Para poder procesar los datos, necesitamos transferirlos de la memoria del equipo anfitrión a la memoria del dispositivo en que serán procesados, y esto se realiza en la línea 95, indicando el destino (`dataDevice`), origen (`dataHost`), tamaño en *bytes* (`sizeof(float)*(N+extra)`), y el tipo de transferencia (`cudaMemcpyHostToDevice`).

```
95     e=cudaMemcpy(dataDevice, dataHost, sizeof(float) * (N+extra),
       cudaMemcpyHostToDevice);
96     if (cudaCheckError(e)) exit(EXIT_FAILURE);
```

Una vez que los datos están en la GPU podemos ejecutar un *kernel* sobre ellos. Elegimos una malla unidimensional con `NB` bloques, y bloques unidimensionales de `TPB` hilos. Con esa configuración de lanzamiento, ejecutamos el *kernel* `functionMin`.

```
99   dim3 grid(NB,1,1);
100  dim3 block(TPB,1,1);
101  functionMin<<<grid, block>>>(dataDevice, minPerBlockDevice,
       minPerBlockIdxDevice, N+extra);
102  e=cudaDeviceSynchronize();
103    if (cudaCheckError(e)) exit(EXIT_FAILURE);
104  e=cudaGetLastError();
105    if (cudaCheckError(e)) exit(EXIT_FAILURE);
```

El *kernel* `functionMin` (l. 11) recibe un apuntador a memoria global que contiene los datos de entrada, dos apuntadores a memoria global de los arreglos donde los resultados serán escritos y una variable en registro para cada hilo que guarda el tamaño total del arreglo. Cada bloque tiene a su disposición dos arreglos de memoria compartida (l. 14 y 15) para realizar lecturas y escrituras eficientes para el trabajo que llevarán a cabo.

```
7  __device__ float f(float x){
8    return (x-1)*(x-1);
9  }

11 __global__ void functionMin(float* array,
12                             float* minPerBlock, int* minPerBlockIdx, int N){

14   __shared__ float minSM[TPB];
15   __shared__ int minIdxSM[TPB];
```

El ciclo `for` que empieza en la línea 17 y termina en la línea 41 tiene por objetivo realizar las repeticiones de cada bloque necesarias para procesar enteramente el arreglo de entrada. Funciona de la siguiente manera: cada hilo tiene un índice `globalIdx` que corresponde a

un único elemento de `array` (se forma con la suma de `threadIdx.x`, el identificador de cada hilo dentro de un bloque, más el producto de `blockDim.x`, el número de hilos por bloque, por `blockIdx.x`, el identificador del bloque dentro de la malla); cada hilo ejecuta el ciclo siempre que `globalIdx<N`, o sea, mientras `globalIdx` no exceda las dimensiones del arreglo; y en cada ejecución `globalIdx` incrementa en `gridDim.x*blockDim.x`, el número total de hilos lanzados en la ejecución. Esta «zancada» se conoce, en inglés, como *stride* y es un mecanismo útil para procesar un arreglo cuyo tamaño excede el número de hilos que se requerirían para procesarlo, se puede extender a arreglos bi o tridimensionales.

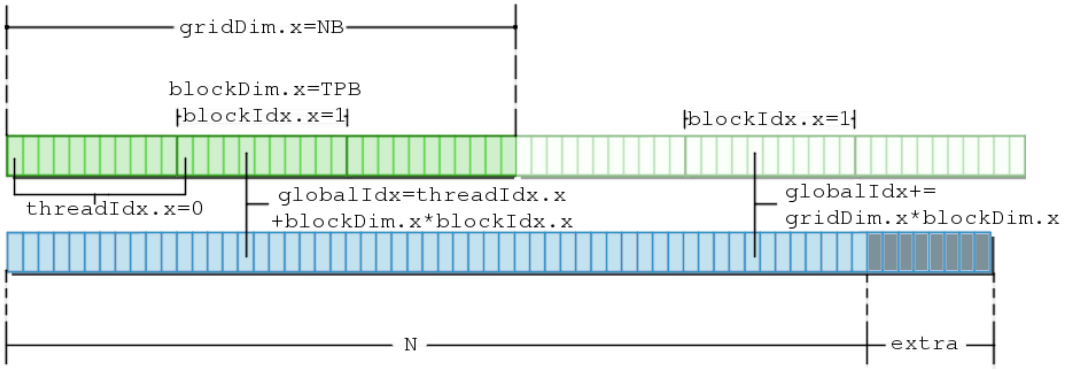


FIGURA 2.11: Ejemplo de uso del *stride* y variables utilizadas para el cálculo de índices.

En la primera parte del ciclo, cada hilo lee el valor que le toca según su `globalIdx`, le aplica la función `f` (ejecutable en GPU y definida en la línea 7 con la palabra reservada `__device__`) y guarda el resultado (l. 19) y el índice (l. 20) en los arreglos de memoria compartida por bloque. Dado que todos los hilos de un mismo bloque procesarán esta información, requerimos que todos los hilos hayan hecho su lectura y cálculo antes de continuar el procesamiento, y para lograrlo usamos sincronización de barrera con la función `__syncthreads()`.

```
17   for(int globalIdx=threadIdx.x+blockDim.x*blockIdx.x;
18           globalIdx<N; globalIdx+=gridDim.x*blockDim.x){
19     minSM[threadIdx.x]=f(array[globalIdx]);
20     minIdxSM[threadIdx.x]=globalIdx;
21     __syncthreads();
```

El ciclo `for` interno que va de la línea 23 a la 33 es la reducción, por bloque, de los elementos del arreglo de memoria compartida que almacenamos previamente. En una primera ejecución, la mitad de los hilos del bloque (l. 23) eligen el mínimo entre el elemento del arreglo `minSM` que le corresponde al hilo en la posición `threadIdx.x` (o sea, un elemento de la primera mitad del arreglo) y el elemento, de ese mismo arreglo, en la

posición `threadIdx.x+activeThreads` (o sea, un elemento de la segunda mitad del arreglo), y lo guardan en su localidad del arreglo compartido, reduciendo un arreglo de tamaño `blockDim.x` a un arreglo de tamaño `blockDim.x/2` (fig. 2.12). El ciclo se repite hasta que no haya hilos activos, y en cada etapa se reduce este número a la mitad.

```
23   for(int activeThreads=blockDim.x>>1;
24             activeThreads>0; activeThreads>>=1){
25       if(threadIdx.x < activeThreads){
26         minSM[threadIdx.x]=fmin(minSM[threadIdx.x],
27                                 minSM[threadIdx.x+activeThreads]);
28         if(minSM[threadIdx.x]==minSM[threadIdx.x+activeThreads]){
29           minIdxSM[threadIdx.x]=minIdxSM[threadIdx.x+activeThreads];
30         }
31       }
32       __syncthreads();
33     }
```

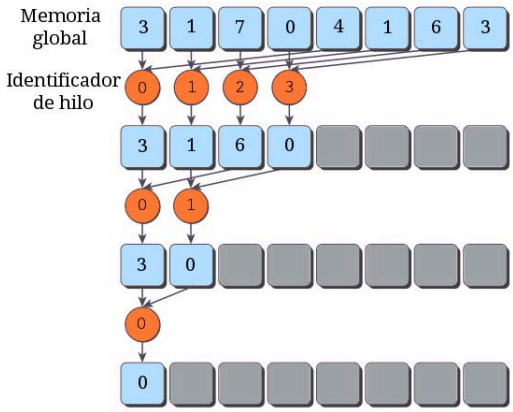


FIGURA 2.12: Ejemplo de reducción de un arreglo en paralelo.

Después de la reducción descrita, el hilo `0` de cada bloque se encarga de escribir, en memoria global, el resultado del procesamiento de ese bloque en los arreglos `minPerBlock`, que contiene el valor mínimo de la función que fue evaluada en GPU; y `minPerBlockIdx`, que contiene la posición del elemento en el arreglo de entrada original que produjo el valor mínimo. Si lanzamos suficientes bloques para cubrir el arreglo, esta asignación se haría fácilmente con `blockIdx.x`, el identificador único de cada bloque dentro de la malla, pero estamos considerando el caso en que los bloques utilizan el *stride* y procesan múltiples secciones de los datos.

```
23 if(threadIdx.x==0){
24       minPerBlock[blockIdx.x+(globalIdx/(gridDim.x*blockDim.x))*gridDim.x]     =
     minSM[threadIdx.x];
25       minPerBlockIdx[blockIdx.x+(globalIdx/(gridDim.x*blockDim.x))*gridDim.x] =
     minIdxSM[threadIdx.x];
```

```
26      }
27    }
28  }
```

Después de analizar el funcionamiento del *kernel* que ejecutamos en la línea 101, regresamos al flujo del programa en la función `main`. Necesitamos copiar los resultados del procesamiento en GPU (el mínimo por bloque de hilos) de vuelta al equipo anfitrión, para terminar la reducción en CPU. Para hacerlo, reservamos espacio en la memoria del equipo anfitrión y copiamos con `cudaMemcpy`, cambiando el origen y destino como es necesario y utilizando la opción `cudaMemcpyDeviceToHost`. Después de esta operación podemos liberar la memoria que los datos ocupaban en la GPU.

```
108   float *minPerBlockHost;
109   int *minPerBlockIdxHost;
110   minPerBlockHost=(float *)malloc(sizeof(float)*(numberOfBlocks));
111   minPerBlockIdxHost=(int *)malloc(sizeof(int)*(numberOfBlocks));
112   e=cudaMemcpy(minPerBlockHost, minPerBlockDevice, sizeof(float) * numberOfBlocks
        , cudaMemcpyDeviceToHost);
113     if (cudaCheckError(e)) exit(EXIT_FAILURE);
114   e=cudaMemcpy(minPerBlockIdxHost, minPerBlockIdxDevice, sizeof(int) *
        numberOfBlocks, cudaMemcpyDeviceToHost);
115     if (cudaCheckError(e)) exit(EXIT_FAILURE);


118   e=cudaFree(minPerBlockDevice); if (cudaCheckError(e)) exit(EXIT_FAILURE);
119   e=cudaFree(minPerBlockIdxDevice); if(cudaCheckError(e)) exit(EXIT_FAILURE);
120   e=cudaFree(dataDevice); if (cudaCheckError(e)) exit(EXIT_FAILURE);
```

Finalmente, reducimos el arreglo en CPU (se recorre el arreglo y se almacena el valor mínimo) e imprimimos el resultado en pantalla.

```
123   float minX;
124   float minFofX;
125   int minIdx;
126   min(minPerBlockHost, minPerBlockIdxHost, &minFofX, &minIdx,   numberOfBlocks);
127   minX=dataHost[minIdx];
128   printf("Minimum value is at idx=%u, x=%f and f(x)=%f\n", minIdx, minX, minFofX)
        ;

130   return 0;
```

Si el ejemplo está en un archivo `ejemplo.cu`, puede compilarse con el compilador de CUDA C. El comando `nvcc ejemplo.cu -o ejemplo` produce un ejecutable de nombre `ejemplo` que realiza lo descrito.

El ejemplo no pretende ser exhaustivo, sino servir como referencia del uso de algunas instrucciones y técnicas. Adaptar un problema al modelo de programación de CUDA siempre es diferente y tiene sus dificultades.

Si el lector encontró interesante la programación en GPU a través de la plataforma CUDA, existen varias referencias que pueden usarse para aprender de forma autodidacta y para ampliar el conocimiento de esta plataforma. Una buena manera de empezar es leyendo y probando los ejemplos de [14], donde paso a paso se ilustran conceptos con ejemplos; y en paralelo, leyendo [15] o [10]. Como referencia rápida se encuentra [1], que es continuamente actualizada con las nuevas características. En [9], y sobre todo en [16], se encuentra un enfoque más detallado y algunos temas avanzados que se recomiendan una vez que el lector se sienta seguro de las bases y haya programado sus primeros programas con CUDA.

Cap´tulo 3

Sensores RGB-D

Los sentidos nos permiten conocer, como humanos, lo que hay alrededor nuestro, y diariamente vemos, tocamos, olemos y oímos nuestro ambiente para tomar decisiones. Si queremos que un aparato artificial sea capaz de realizar actividades como las que nosotros podemos desempeñar, deberíamos concederle, al menos, algunos de esos sentidos para que tal ente tenga una oportunidad en un ambiente cambiante. Los avances tecnológicos y la reducción de costos en producción de tecnología han logrado que existan sensores de precio accesible que dan a la computadora la oportunidad de percibir, hasta cierto punto, el ambiente.

Yo mismo he sido testigo de estos avances y lo seguiré siendo. Para mí, por ejemplo, ha sido natural, desde niño[1], tener un micrófono y conectarlo a la computadora para que sea capaz de registrar sonido. Para algunos sobrinos míos hoy es natural darle una instrucción, por medio de voz, al celular. En cambio, mi octogenaria abuela me cuenta que jamás imaginó en su niñez algo de esto. Yo, de niño, jamás creí que tener un aparato que le dejara saber a la computadora qué tan lejos está un objeto sería tan fácil. A los niños de hoy, no obstante, les parece natural tener un aparato como el *Kinect* de Microsoft [17], y seguramente los niños en algunos años encontrarán normal que una computadora o algún robot sea capaz de reconocer objetos sin dificultad. Es probable que cuando yo sea un anciano me quede, como dice mi abuela, con el ojo cuadrado con los avances y lo que las computadoras serán capaces de percibir y hacer, y mis nietos encuentren todo eso muy normal.

Este capítulo explora brevemente algunos sensores que permiten adquirir información geométrica de una escena. Esta información se almacena en un arreglo matricial que llamamos mapa de profundidad. Para definirlo y entenderlo mejor, pensemos en una imagen

[1] Quien escribe esto tuvo, afortunadamente, una infancia feliz. Me atrevo a suponer que la del lector no estuvo mal o que, si lo fue, tuvo la fortaleza para llegar a este punto. Hagamos lo que podamos para que más ninos puedan tener infancias similares y no tengan que preocuparse por la calamidad cotidiana del planeta.

$I(x, y)$ como un arreglo rectangular en el que cada elemento (x, y) —llamado pixel— contiene información del color de la superficie que representa ese pixel. Entendamos ahora como mapa de profundidad, un arreglo rectangular $D(x, y)$ en el que cada elemento no representa el color, sino la distancia del sensor al área representada por el elemento (x, y), (fig. 3.1). Un sensor RGB-D es un dispositivo que captura y proporciona información del color y de la profundidad de un ambiente.

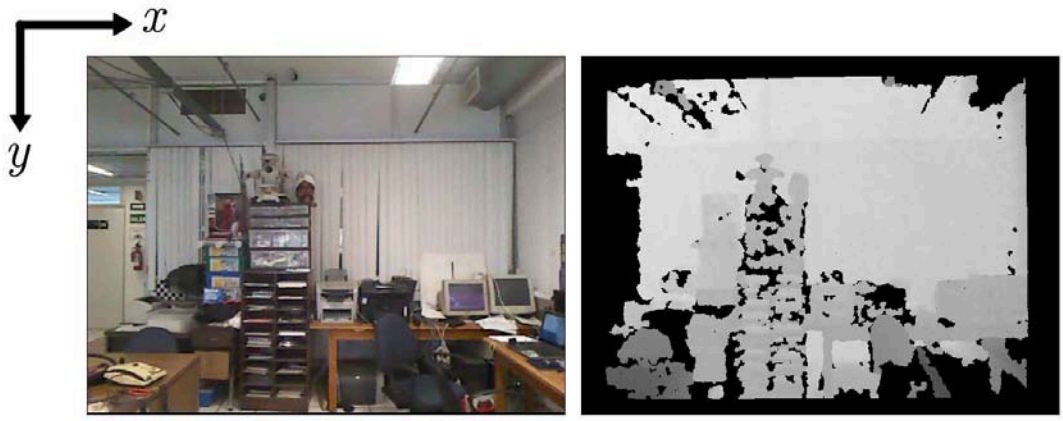


FIGURA 3.1: En cada pixel, la imagen guarda un color, y el mapa de profundidad guarda una distancia.

Independientemente de la manera en que cada sensor adquiere y transforma la información de profundidad de una escena, esperamos que el mapa de profundidad provea una representación fidedigna de la realidad. Conocer las diferentes maneras de adquisición puede ayudarnos a comprender y atacar las deficiencias del mapa de profundidad que se obtiene con cada dispositivo.

Daremos un vistazo rápido a tres tipos de sistemas: las cámaras de tiempo de vuelo, los sistemas estereoscópicos de visión y las cámaras basadas en luz estructurada. Dentro de esta última categoría se encuentra el Kinect de Microsoft, que es el dispositivo utilizado para obtener los mapas de profundidad en esta tesis. Al final del capítulo se presenta un modelo de cámara y una metodología para convertir un mapa de profundidad a puntos en un sistema coordenado de referencia.

3.1. Cámaras de tiempo de vuelo (*Time-of-Flight range cameras*)

Un sensor puntual de tipo *ToF* estima la distancia radial a un punto de una escena utilizando el principio de tiempo de vuelo o RADAR (*Radio Detection and Ranging*). De una forma

muy simplificada, el sensor estima la distancia d que recorre una onda a la velocidad de la luz c en un tiempo t. Suponiendo que el emisor de la onda y el receptor de la misma están alineados, un rayo emitido viajará hacia un punto de la escena, recorriendo una distancia d, rebotará y recorrerá, nuevamente pero en dirección al receptor, una distancia d, todo esto en un tiempo t, por lo que la distancia del sensor al punto de la escena es $d = \frac{ct}{2}$. (Fig. 3.2).

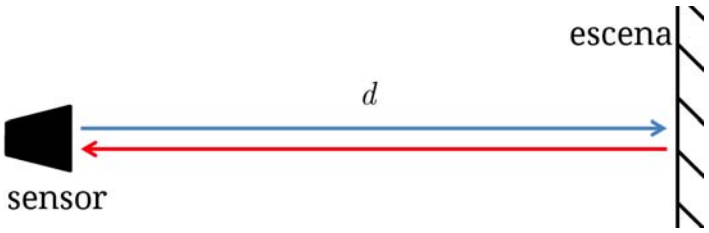


FIGURA 3.2: El rayo recorre una distancia d hasta rebotar en la escena. Y recorre la misma distancia para regresar al sensor.

Para medir áreas mayores, el sensor puede ser montado en algún mecanismo que permita desplazarlo, por ejemplo, en una línea. También puede crearse un arreglo de ellos. La segunda opción se conoce como una cámara matricial *ToF* y puede adquirir información de diferentes puntos de una escena de manera simultánea. Algunas marcas comunes son MESA Imaging, PMD Technologies y Optrima SoftKinect. La segunda versión del Kinect de Microsoft utiliza una cámara de este tipo [18], en lugar del patrón de luz estructurada del que hablaremos más adelante.

Como seguramente se imagina el lector, la manera de obtener la distancia, en la práctica, no es tan fácil como la descrita en el primer párrafo, ni el arreglo de sensores es tan ideal como podría dar a entender el segundo. En el primer caso, las cámaras más comunes trabajan con una señal infrarroja sinusoidal modulada y se conocen como cámaras de onda continua (*Continous Wave ToF Camera*). Cuando la señal rebota en la escena, se dispersa y no regresa con la misma intensidad, además de ser afectada por el ruido del ambiente. Obtener una señal sinusoidal perfecta es, en sí, un problema y puede introducir errores en la medición. En cuanto al arreglo, en la práctica se utiliza un arreglo de emisores cuya información es recolectada por un arreglo de sensores como en la figura 3.3. En estas disposiciones, el lente que atraviesan los rayos reflejados en la escena antes de llegar al sensor también juega un papel importante. En [19] puede encontrarse una explicación detallada del funcionamiento de estas cámaras.

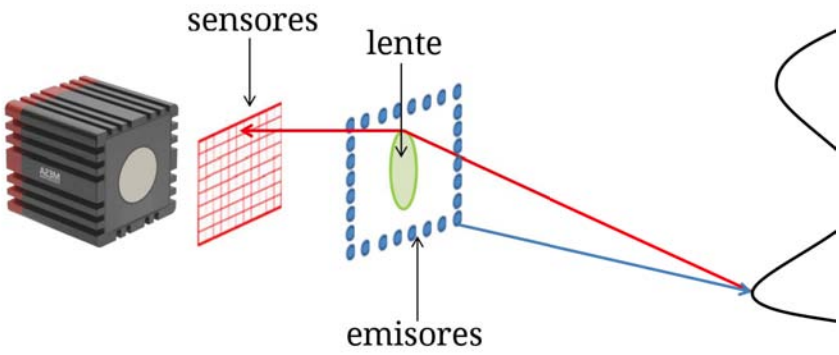


FIGURA 3.3: Ejemplo del funcionamiento de una cámara ToF. [19]

3.2. Sistemas estereoscópicos de visión

Un sistema de visión estéreo está formado por dos cámaras normales (usualmente iguales) que capturan la misma escena. Estas cámaras pueden calibrarse y rectificarse de manera que el sistema sea equivalente a tener dos cámaras idénticas con sensores coplanares y alineados, además de tener ejes ópticos paralelos. También existen productos comerciales con dos cámaras integradas como la *ZED* de la compañía *STEREOLABS*.

En un sistema mínimo, como el de la figura 3.4, cada cámara tiene un sistema tridimensional (x_I, y_I, z_I) (para la cámara izquierda) para referir los puntos de la escena, y uno bidimensional (u_I, v_I) para los pixeles de la proyección. El sistema de la cámara izquierda suele utilizarse como referencia de los puntos de una escena. Tomando esto en cuenta, un punto $\mathbf{P} = (x, y, z)$ de la escena se proyecta a los pixeles p_I y p_D en las cámaras izquierda y derecha, respectivamente, con coordenadas $p_I = (u_I, v_I)$ y $p_I = (u_D = u_I - d, v_D = v_I)$, como se muestra en la figura 3.5, dando lugar a la disparidad entre las dos imágenes $d = u_I - u_D$. Esta disparidad es inversamente proporcional al valor de profundidad z según la relación $z = \frac{bf}{d}$, donde b es la línea base entre los orígenes de los sistemas tridimensionales de las cámaras izquierda y derecha.

Los pixeles p_I y p_D se llaman conjugados, y encontrarlos en las dos imágenes es una de las partes complicadas de los sistemas de visión estereoscópica. Se conoce como problema de correspondencia, y existen enfoques globales, que trabajan sobre toda la imagen, y locales, que trabajan sobre una vecindad de la imagen derecha para buscar el correspondiente a p_I. En [20] pueden encontrarse algunos métodos para hallar pixeles correspondientes.

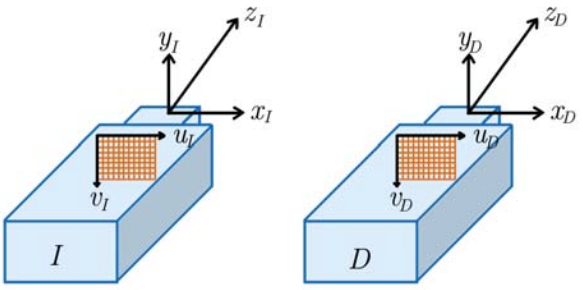


FIGURA 3.4: Sistemas de referencia de un sistema de visión.

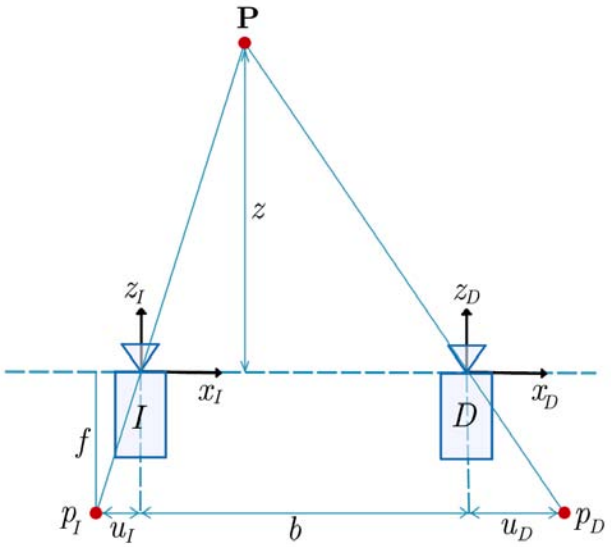


FIGURA 3.5: Obtención de la profundidad en un sistema de visión.

3.3. Cámaras basadas en luz estructurada

En una pared lisa y de color uniforme, encontrar pixeles correspondientes entre dos imágenes de un sistema como el de la sección anterior es prácticamente imposible. Los sistemas que utilizan una luz estructurada permiten detectar pixeles correspondientes en este caso.

En una configuración como la de la figura 3.6, en donde tenemos una cámara de referencia y un proyector del patrón de luz, un pixel p_L del patrón de luz es proyectado sobre un punto $\mathbf{P} = (x, y, z)^\intercal$ en el sistema de referencia de la cámara y es capturado en el sistema de referencia bidimensional de la imagen de la cámara en un pixel p_C. Entre p_C, $\mathbf{P}$ y p_L se forma un triángulo como el de la figura 3.5 y la profundidad z puede estimarse como se discutió en la sección anterior.

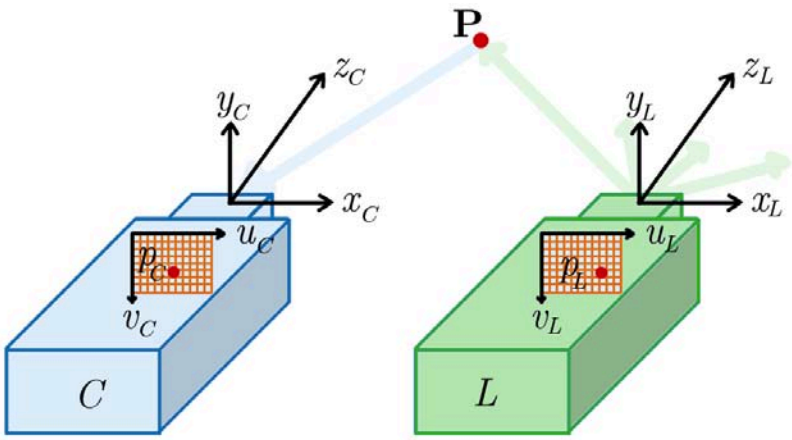


FIGURA 3.6: Sistema de visión con luz estructurada.

La función del patrón de luz es permitir la detección de correspondencias entre un punto proyectado y uno capturado, mediante la deformación del patrón de luz. La estructura del patrón juega un papel importante y se diseña meticulosamente para estimar la disparidad de forma rápida y a prueba de errores. En [19] se encuentra una descripción detallada y más referencias del diseño de estos patrones.

En el proceso de proyección y adquisición existen varios factores que pueden interferir con la calidad del mapa de profundidad. Por ejemplo, si el patrón es proyectado sobre alguna superficie no reflectante o en una que refleja demasiada luz, este se pierde y la distancia no puede ser estimada. La iluminación externa también es un factor que puede producir fallas, puesto que los proyectores suelen trabajar con rayos infrarrojos. La oclusión es otra causa de pixeles que son percibidos por la cámara pero que no pueden asociarse a un punto del patrón (porque algo obstruye el camino entre el proyector y el punto de la escena).

3.3.1. Microsoft Kinect

Cuando el Kinect de Microsoft se introdujo al mercado en noviembre de 2010, como dispositivo de interacción natural del Xbox 360, se vendieron ocho millones de dispositivos en los primeros sesenta días de su lanzamiento, superando productos como el *iPad* (tres millones) de *Apple*. La alta disponibilidad y precio accesible despertaron la curiosidad de los desarrolladores e investigadores. Los primeros controladores para utilizar el Kinect con una computadora fueron producto de una recompensa de la compañía *Adafruit Industries*. Sin embargo, la compañía *PrimeSense* (encargada del desarrollo del dispositivo), al ver el interés de la comunidad por utilizar estos dispositivos, liberó controladores y un marco de

trabajo (OpenNI) compatibles con Kinect y con aparatos similares, como el Xtion de ASUS. Microsoft cuenta con el Kinect SDK (*Software Development Kit*) y con controladores oficiales. Hoy puede utilizarse el Kinect en una variedad de plataformas y sistemas operativos, y es una opción accesible de interfaz con la computadora o como sensor de profundidad.

El sensor Kinect (fig. 3.7) tiene una cámara RGB que trabaja a 30 Hz y con una resolución de 640x480 pixeles (también puede operar a 10-15 Hz con resolución de 1280x1024) y 8 *bits* por canal.

FIGURA 3.7: Sensor Kinect.

Al ser un dispositivo que trabaja con luz estructurada, cuenta con un emisor de un patrón de luz infrarroja de 830 nm (fig. 3.8) y una cámara infrarroja que captura esta frecuencia y la proyección de este patrón sobre la escena. El sistema tiene un ángulo de visión horizontal de 58 grados, vertical de 42 y diagonal de 70. El rango de operación está entre los 0.8 m y los 4 m. El mapa de profundidad, con una resolución de 320x240 (o escalada a 640x480) y con una cuantización de 11 *bits*, se calcula, a partir de la imagen capturada por la cámara infrarroja, en el circuito integrado PS1080 de la compañía PrimeSense que está integrado en el Kinect.

Algunos controladores y bibliotecas, como *freenect*, pueden proveer la información del mapa de profundidad en esa cuantización (11 *bits*), como un entero entre 0 y 2047. Para convertir ese número a metros puede utilizarse una fórmula (como la sugerida en [21]) que tome en cuenta que estos enteros no se relacionan de manera lineal con la distancia. Pero en general, los marcos de trabajo como OpenNI y Kinect SDK proporcionan funciones para obtener el mapa de profundidad en milímetros en un arreglo de enteros de 16 *bits*.

3.4. Modelo de cámara estenopeica

El modelo de cámara estenopeica (*pinhole*, en inglés) nos permite establecer la relación entre los pixeles de una imagen y los puntos de una escena. Particularmente, entre un pixel

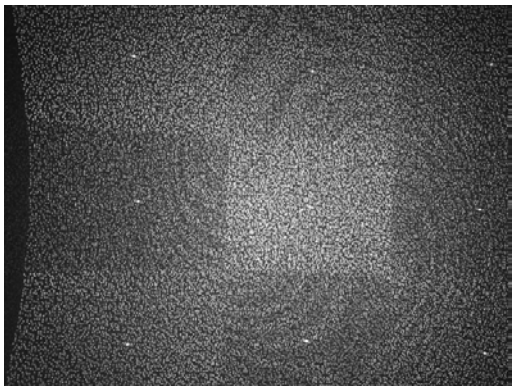


FIGURA 3.8: Patrón infrarrojo del sensor Kinect. [22]

de un mapa de profundidad y el punto en el espacio que representa.

Pensemos en un sistema de referencia tridimensional, que llamaremos sistema de coordenadas de la cámara, con ejes x, y y z, que siguen la convención de la mano derecha, con origen en un punto O llamado centro de proyección; y en un plano, que llamaremos plano de la imagen o sensor, paralelo al plano xy, y que se cruza con el eje z a una distancia f. En este plano tenemos un sistema de referencia bidimensional, x', y', llamado sistema coordenado de la imagen, con origen en el punto en que el plano se interseca con el eje z (fig. 3.9).

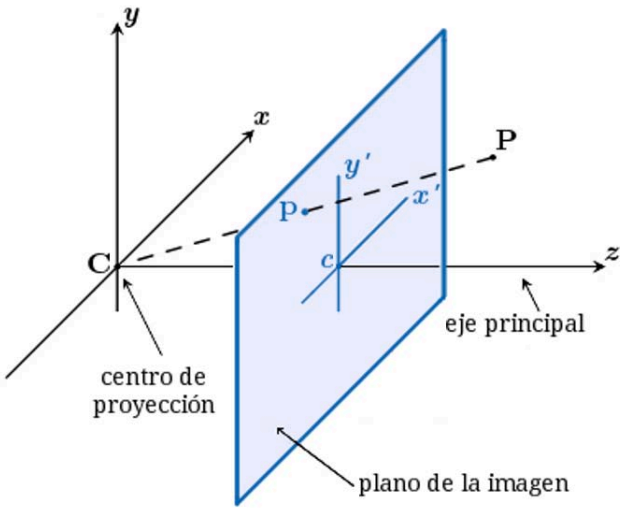


FIGURA 3.9: Modelo de cámara estenopeica.

En este modelo, los puntos $\mathbf{P} = (x, y, z)^\intercal$ de una escena tridimensional se proyectan hacia el centro de proyección O. Un pixel $\mathbf{p} = (x', y')^\intercal$ se obtiene de la intersección de la

línea que pasa por **P** y O con el plano de la imagen.

La relación entre **P** y **p** puede determinarse fácilmente mediante semejanza de triángulos (fig. 3.10).

$$\begin{cases} x' = f\dfrac{x}{z} \\ y' = f\dfrac{y}{z} \end{cases} \tag{3.1}$$

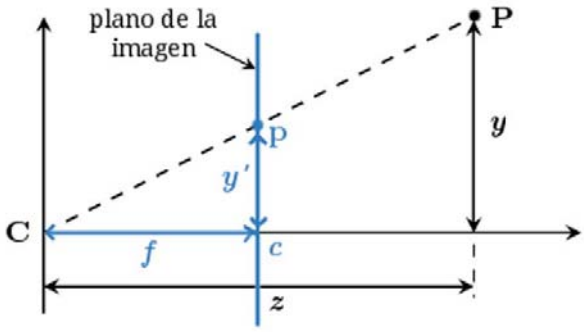


FIGURA 3.10: Relación entre y y y'.

Las ecuaciones 3.1 pueden escribirse en forma matricial, y considerando que el centro de la imagen puede estar en cualquier punto (c_x, c_y) del plano de la imagen, como sigue:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} f\frac{x}{z} + c_x \\ f\frac{y}{z} + c_y \end{pmatrix} \tag{3.2}$$

Estas ecuaciones son útiles cuando conocemos la longitud focal, la posición de los puntos de intersección y el centro de la imagen en unidades de longitud (milímetros, por ejemplo). Pero usualmente trabajamos con pixeles como unidad, por lo que resulta conveniente tener una formulación en estas unidades. En las siguientes dos subsecciones damos un vistazo rápido a las coordenadas homogéneas y a los parámetros intrínsecos y extrínsicos de una cámara, que en conjunto permiten obtener la formulación deseada.

3.4.1. Coordenadas homogéneas

En geometría proyectiva es común asociarle a un punto $\mathbf{p} = (x', y')^\intercal$ del plano cartesiano una representación en coordenadas homogéneas tridimensionales $\tilde{\mathbf{p}} = (\lambda x', \lambda y', \lambda)^\intercal$ con $\lambda \in \mathbb{R}$. Dada una representación en coordenadas homogéneas $\tilde{\mathbf{p}}$, el punto $\mathbf{p}$ puede obtenerse dividiendo las dos primeras coordenadas de $\tilde{\mathbf{p}}$ entre la tercera coordenada (siempre que no sea cero). Geométricamente, podemos entender el conjunto de coordenadas homogéneas de

un punto bidimensional como la recta que pasa por el origen de un sistema tridimensional, que tiene por ecuación $L(\lambda) = \lambda(x', y', 1)$, y que se interseca con el plano $z = 1$ en $(x', y', 1)$.

Un punto $\mathbf{P} = (x, y, z)^\intercal$ en tres dimensiones cartesianas también puede ser representado en coordenadas homogéneas como $\tilde{\mathbf{P}} = (\lambda x, \lambda y, \lambda z, \lambda)^\intercal$.

Esta representación permite, entre otras cosas, expresar relaciones no lineales, como la ecuación 3.2, como el producto de matrices:

$$z \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{3.3}$$

3.4.2. Parámetros intr´nsecos y extr´nsecos de una cámara

Las imágenes se capturan utilizando un arreglo de dispositivos (con tecnología CCD o CMOS) que convierten la luz a electrones. Estos sensores producen la información de elementos discretos llamados pixeles. Hasta ahora hemos considerando las cantidades f, c_x y c_y en unidades de longitud, pero resulta más conveniente trabajar con pixeles, por lo que introducimos las constantes k_x y k_y como los pixeles por unidad de longitud (pixeles/unidad de longitud) en los ejes x y y.

Con estas constantes podemos escribir la ecuación 3.7 como:

$$z \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \mathbf{K} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} f_x & 0 & c'_x \\ 0 & f_y & c'_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{3.4}$$

con $f_x = k_x f$, $f_y = k_y f$, $c'_x = k_x c_x$ y $c'_y = k_y c_y$. Estos parámetros se llaman intrínsecos y la matriz $\mathbf{K}$ suele llamarse matriz de parámetros intrínsecos, y se obtiene mediante un método de calibración. En el capítulo sexto de [23] y en el cuarto de [19] se describen algunos métodos de calibración. Existen herramientas, como [24], que permiten obtener los parámetros intrínsecos del sensor de profundidad y de la cámara RGB de un Kinect.

En muchas ocasiones, los puntos estarán referidos en un sistema de coordenadas diferente al de la cámara, que es el que hemos contemplado hasta ahora. Este sistema suele llamarse sistema de coordenadas de mundo. La relación entre el sistema de la cámara y el de mundo se puede representar con una matriz de rotación $\mathtt{R}$ y un vector de traslación $\mathbf{t}$, llamados parámetros extrínsecos, que representen la relación entre esos dos sistemas, de manera que el equivalente de un punto $\mathbf{P}_m = (x_m, y_m, z_m)^\intercal$ en coordenadas de mundo es un punto $\mathbf{P}$ en coordenadas de cámara dado por:

$$\mathbf{P} = \mathtt{R}\mathbf{P}_m + \mathbf{t} \tag{3.5}$$

Al expresar el punto $\mathbf{P}_m$ en coordenadas homogéneas como $\tilde{\mathbf{P}}_m = (x_m, y_m, z_m, 1)^\intercal$ y combinar las ecuaciones 3.4 y 3.5, podemos obtener una expresión que asocia los puntos en un sistema de coordenadas de mundo con los pixeles de una proyección:

$$z\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \mathbf{K}(\mathtt{R}\mathbf{t})\tilde{\mathbf{P}}_m = \begin{pmatrix} f_x & 0 & c'_x \\ 0 & f_y & c'_y \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{21} & r_{22} & r_{23} & t_2 \end{pmatrix}\begin{pmatrix} x_m \\ y_m \\ z_m \\ 1 \end{pmatrix} \tag{3.6}$$

3.4.3. Conversión de un mapa de profundidad a coordenadas de mundo

Bajo el modelo de la cámara estenopeica y tomando la ecuación 3.4 podemos, fácilmente, determinar las coordenadas de mundo de un punto $\mathbf{P}_m = (x_m, y_m, z_m)^\intercal$ en una escena a partir de su posición (x', y') en el mapa de profundidad[2] y de la medición de profundidad z.

Las ecuaciones 3.7, obtenidas a partir de 3.4, nos dan las coordenadas del punto en coordenadas de la cámara, por lo que solamente hace falta multiplicar por una matriz T_m que represente la transformación del sistema de la cámara al sistema del mundo (Eq. 3.8).

$$\begin{cases} x = \dfrac{(x' - c'_x)z}{f_x} \\ y = \dfrac{(y' - c'_y)z}{f_y} \\ z = z \end{cases} \tag{3.7}$$

$$\begin{pmatrix} x_m \\ y_m \\ z_m \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{3.8}$$

[2]Estas coordenadas se miden desde la esquina superior izquierda del mapa, y son positivas en x' hacia la derecha y en y' hacia abajo.

Cap´tulo 4

Estado actual

La reconstrucción de superficies es un tema bien estudiado en computación gráfica y en visión por computadora, y tiene aplicaciones en la medicina, la arqueología, la manufactura, y el entretenimiento, por nombrar algunas áreas. Para esta tesis nos interesa el problema de construir, a partir de múltiples medidas de profundidad de una escena o un objeto, una representación tridimensional que se asemeje a la geometría de la realidad. Tales medidas de profundidad pueden obtenerse por medio de alguno de los métodos mencionados en el capítulo anterior. La disponibilidad de dispositivos como Kinect y de unidades de procesamiento paralelo como las GPU provocó un interés en la comunidad para crear, utilizando ambas tecnologías, sistemas de reconstrucción en tiempo real.

En este capítulo se presenta una reinterpretación del sistema KinectFusion como está descrito en [25, 26] esperando que sea de utilidad para quienes, en el futuro, quieran continuar con este trabajo. Se analiza porque, además de ser uno de los primeros que utilizó el Kinect y una GPU para realizar la reconstrucción en tiempo real de una escena, permite comprender mejor los trabajos posteriores y las dificultades que cada uno ataca, que mencionaremos más adelante.

4.1. KinectFusion

El propósito de este sistema es lograr que una persona con un Kinect se mueva libremente por una escena y pueda generar la representación tridimensional de esa escena. Para lograrlo, el sistema rastrea, continuamente, la posición de una cámara con seis grados de libertad ($6DOF$) y fusiona, en tiempo real, la información de los mapas de profundidad provistos por el sensor en un solo modelo global tridimensional. Cada movimiento del usuario proporciona nueva información y ayuda a refinar el modelo. El sistema está formado por las siguientes etapas:

- **Medición de la super cie**. El proceso de filtrar un mapa de profundidad y obtener, a partir de él, los puntos y vectores normales (a la superficie en ese punto) de cada pixel del mapa.
- **Actualización del modelo.** Fusionar la distancia medida en ese momento, con el modelo global —almacenado en una matriz tridimensional como una función de distancia con signo truncada *truncated signed distance function, TSDF*)—, tomando en cuenta la posición estimada del sensor.
- **Predicción de la super cie.** En esta etapa se obtienen, mediante trazado de rayos sobre el modelo global, los puntos y vectores normales que se utilizarán para estimar la posición del sensor.
- **Estimación de la posición y orientación del sensor.** Mediante un algoritmo *Iterative Closest Points* (ICP) se obtiene la transformación que representa la posición y orientación del sensor en coordenadas globales desde el cual se capturó el cuadro nuevo. El algoritmo trabaja a escalas múltiples y compara los puntos y vectores normales del mapa de profundidad entrante con aquellos de la superficie predicha, descrita en la etapa anterior.

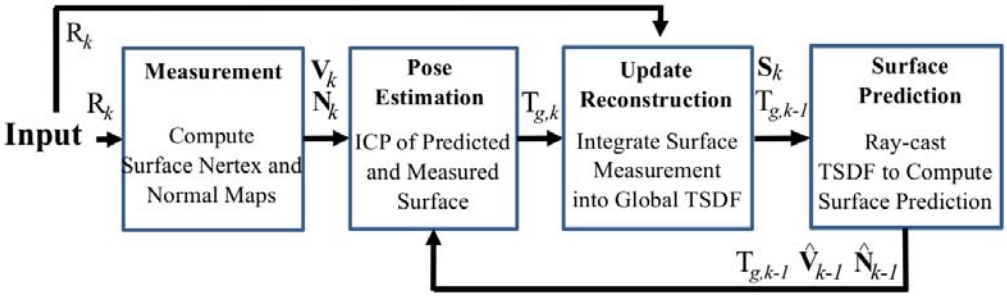


FIGURA 4.1: El sistema KinectFusion como es presentado en [25].

El flujo de estas etapas se muestra en la figura 4.1. Para comprenderla mejor, hay que tomar en cuenta que, en un tiempo k, R_k representa el mapa de profundidad sin alteraciones. $\mathbf{V}_k$ y $\mathbf{N}_k$ la posición de cada punto y los vectores normales obtenidos en la primera etapa, respectivamente. La transformación que convierte las coordenadas de cámara a las coordenadas globales g está dada por $\mathrm{T}_{g\,k}$, y la correspondiente al tiempo anterior se denota con $\mathrm{T}_{g\,k-1}$. Los puntos $\mathbf{V}_{k-1}$ y los vectores normales $\mathbf{N}_{k-1}$ se obtienen al trazar rayos sobre el modelo completo desde la posición del tiempo anterior $k-1$, y se utilizan,

en conjunto con los vectores correspondientes del tiempo actual k, para obtener la posición estimada del sensor, representada por $\mathrm{T}_{g\,k}$.

4.1.1. Medición de la super cie

En esta primera etapa, los valores del mapa $\mathrm{R}_k(\mathbf{u})$, que proporcionan una medida de la profundidad en cada pixel $\mathbf{u} = (u, v)^\intercal$ en el dominio de la imagen $\mathbf{u} \in \mathcal{U} \quad \mathbb{R}^2$, son transformados, mediante un filtro bilateral[27] (fig. 4.2), a un nuevo mapa de profundidad $\mathrm{D}_k(\mathbf{u})$ con ruido reducido pero preservando los bordes:

$$\mathrm{D}_k(\mathbf{u}) = \frac{1}{W_p} \sum_{\mathbf{q}\ \mathcal{U}} \mathcal{N}_{\ s}(\ \mathbf{u} - \mathbf{q}\ _2)\mathcal{N}_{\ r}(\ \mathrm{R}_k(\mathbf{u}) - \mathrm{R}_k(\mathbf{q})\ _2)\mathrm{R}_k(\mathbf{q}) \tag{4.1}$$

donde $\mathcal{N}\ (t) = \exp(-t^2\ ^{-2})$, W_p es una constante de normalización y $\mathbf{x}\ _2$ es la norma euclidiana de un vector $\mathbf{x}$.

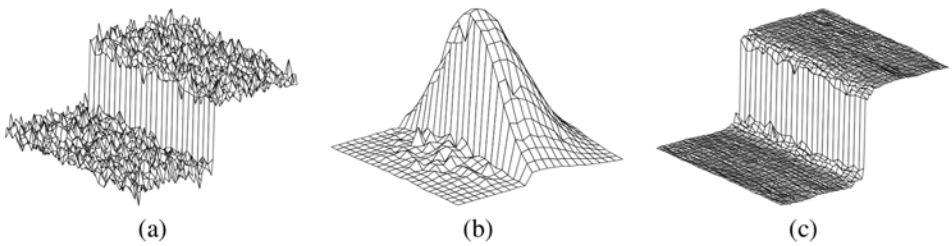


FIGURA 4.2: (a) Ejemplo de una superficie ruidosa a la que se le aplica el filtro. (b) La forma del filtro para un punto cerca del borde. (c) Resultado. [27].

Después, los valores filtrados del mapa de profundidad se transforman al sistema de referencia del sensor en el tiempo k para obtener un mapa de puntos $\mathbf{V}_k$, utilizando una matriz de parámetros intrínsecos K y $\mathbf{u}$ en coordenadas homogéneas como $\mathbf{u} = (u, v, 1)^\intercal$, según discutimos en el capítulo anterior.

$$\mathbf{V}_k(\mathbf{u}) = \mathrm{D}_k(\mathbf{u})\mathrm{K}^{-1}\mathbf{u} \tag{4.2}$$

La ecuación 4.2 es un proceso equivalente a utilizar las ecuaciones 3.7 que obtuvimos en el capítulo anterior.

A partir del mapa de puntos se calcula el vector normal $\mathbf{N}_k(u, v)$ a la superficie que se forma entre puntos vecinos de $\mathbf{V}_k(u, v)$. Para obtener cada vector normal $\mathbf{N}_k(u, v)$ se utiliza el producto vectorial de un vector que va del punto $\mathbf{V}_k(u, v)$ al punto adyacente a

la derecha en el mapa $\mathbf{V}_k(u+1,v)$, por otro vector que va de del punto $\mathbf{V}_k(u,v)$ al punto adyacente abajo $\mathbf{V}_k(u,v+1)$. O sea,

$$\mathbf{N}_k(\mathbf{u}) = p[(\mathbf{V}_k(u+1,v) - \mathbf{V}_k(u,v)) \times (\mathbf{V}_k(u,v+1) - \mathbf{V}_k(u,v))] \tag{4.3}$$

donde $p[\mathbf{x}] = \mathbf{x} \quad \mathbf{x}_{\ 2}$.

Además de los mapas de puntos y de vectores normales, se define una máscara $\mathbf{M}_k(u,v)$ que almacena un 1 si la medición de profundidad fue correcta y generó un punto y vector normal correctos, y un 0 de otro modo.

La etapa de estimación de la posición del sensor trabaja con mapas de profundidad, de puntos, y de vectores normales a tres escalas diferentes. El primer nivel está compuesto por los tres mapas discutidos hasta ahora. El segundo nivel del mapa de profundidad D_k^2 se forma aplicando un filtro de bloque sobre el mapa $\mathrm{D}_k(\mathbf{u})$ y tomando muestras para reducir la resolución a la mitad, a partir de este nuevo mapa se calculan los mapas de puntos y de vectores normales con las ecuaciones 4.2 y 4.3. El tercer nivel D_k^3 se forma de manera similar, pero a partir del mapa de profundidad D_k^2.

Obsérvese en este punto que conociendo la matriz $\mathrm{T}_{g\,k}$, formada por una rotación $\mathtt{R}_{g\,k}$ que representa la orientación del sensor y un vector $\mathbf{t}_{g\,k}$ que representa la posición del mismo dentro del sistema de referencia global en el tiempo k, se calcula la posición de un punto en el sistema global $\mathbf{V}_k^g(\mathbf{u}) = \mathrm{T}_{g\,k}\mathbf{V}_k(\mathbf{u})$ (utilizando coordenadas homogéneas, como describimos en la última sección del capítulo anterior) y la orientación del vector normal con $\mathbf{N}_k^g(\mathbf{u}) = \mathtt{R}_{g\,k}\mathbf{N}_k(\mathbf{u})$.

4.1.2. Actualización del modelo

Cada cuadro del mapa de profundidad proporciona nueva información del ambiente capturado por el sensor. Esta nueva información se integra a una representación única como una función de distancia con signo truncada (*truncated signed distance function, TSDF*) con muestras almacenadas en un arreglo tridimensional, basada en [28].

Para comprender mejor la función de distancia con signo truncada, recordemos, primero, cómo representar superficies de manera implícita utilizando una función; después, qué es una función de distancia con signo y, finalmente, en qué consiste la *TSDF*.

Decimos que representamos una interfaz de forma implícita con una función $F(\mathbf{p})$ cuando para todos los puntos $\mathbf{p} \in \mathbb{R}^n$ de la interfaz se cumple que $F(\mathbf{p}) = k$. Usualmente se define un valor $k = 0$ y se dice que la interfaz está definida por el conjunto de nivel $F(\mathbf{p}) = 0$. Por ejemplo, con la función $F(x,y) = x^2 + y^2 - 4$ podemos definir los puntos del círculo con centro en el origen de un sistema cartesiano y de radio cuatro mediante el conjunto de nivel

$F(x, y) = 0$. De esta forma, todos los puntos del círculo pertenecen al conjunto de nivel, los puntos con $F(x, y) > 0$ están afuera del círculo, y los puntos con $F(x, y) < 0$ están dentro del círculo. Una completa introducción a la representación de interfaces mediante funciones implícitas puede encontrarse en [29].

Una función de distancia con signo es una manera de representar una superficie de manera implícita. El valor de la función en cada punto es la distancia al punto más cercano de la interfaz, y toma valores positivos que incrementan desde la superficie y hacia el espacio libre observado; y valores negativos que decrementan su valor desde la superficie y hacia el espacio no observado.

La función de distancia con signo truncada que se utiliza en el sistema KinectFusion, almacena únicamente los valores de distancia cercanos a la interfaz y trunca el resto a un valor , para el espacio observado, y a un valor − , para el no observado (Fig. 4.3).

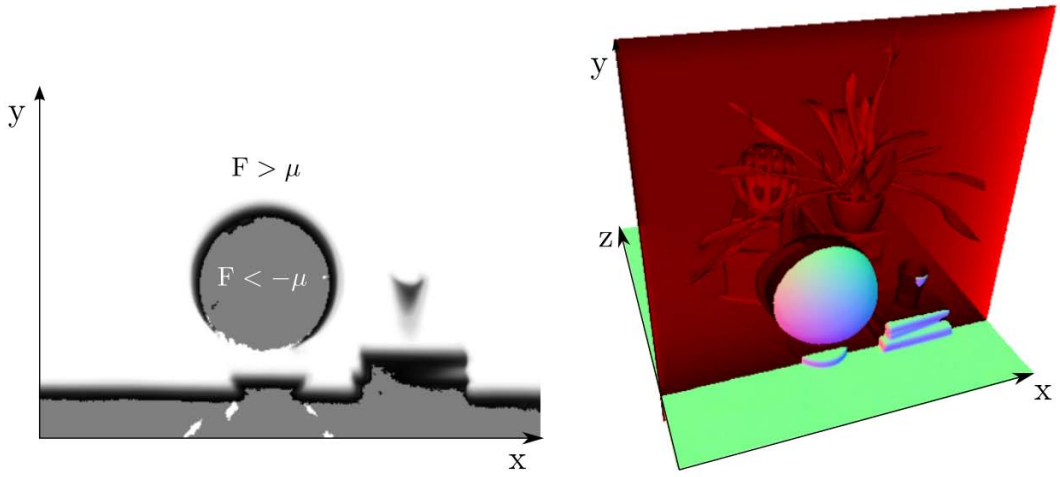


FIGURA 4.3: Un corte sobre la estructura que almacena la TSDF y los valores de la función. En blanco, se muestran los valores $F >$; en gris, los valores $F <$ − ; y cerca de la interfaz, la función de distancia con signo. [25].

En [25] se representa la TSDF que ha integrado los cuadros $1, 2, ..., k$ con $\mathbf{S}_k(\mathbf{p})$, donde $\mathbf{p} \in \mathbb{R}^3$. En la computadora, en realidad, se almacena una discretización de esta función continua en un arreglo tridimensional de dimensiones predefinidas, y suponen que existe una función biyectiva entre los *voxels* (elementos del arreglo tridimensional) y la representación continua; o sea, que para un elemento de la representación continua se puede obtener el voxel asociado y viceversa. Se almacenan dos componentes por cada localidad de la TSDF, a saber, el valor de la función truncada $F_k(\mathbf{p})$ y un peso $W_k(\mathbf{p})$,

$$\mathbf{S}_k(\mathbf{p}) \quad [F_k(\mathbf{p}), W_k(\mathbf{p})]. \tag{4.4}$$

En este sistema, como ya mencionamos, la función $F_k(\mathbf{p})$ almacena la TSDF en el punto $\mathbf{p}$, guardando una función de distancia con signo en los puntos adyacentes a la superficie y truncando ese valor a y − para puntos observados y no observados, respectivamente. Para obtener el valor de F, se utiliza una función de distancia con signo proyectiva que puede paralelizarse. Para un mapa de profundidad sin alteraciones R_k y una estimación de la posición y orientación $\mathrm{T}_{g\,k}$, se puede calcular el valor de la función F_{R_k} como sigue:

$$F_{R_k}(\mathbf{p}) = \frac{1}{\lambda} \mathbf{t}_{g\,k} - \mathbf{p}_{\ 2} - \mathrm{R}_k(\mathbf{x}) \tag{4.5}$$

$$\lambda = \mathrm{K}^{-1}\mathbf{x}_{\ 2} \tag{4.6}$$

$$\mathbf{x} = \mathrm{K}\mathrm{T}_{g\,k}^{-1}\mathbf{p} \tag{4.7}$$

$$(\ ) = \begin{cases} \text{mín} \ \ 1, - \ \ \text{sgn}(\ ) & \text{sii} \quad - \\ \textit{nulo} & \text{en otro caso} \end{cases} \tag{4.8}$$

La ecuación 4.8 realiza el proceso de truncamiento. Conserva el signo de , el argumento de , y para valores de este argumento mayores o iguales que − almacena el mínimo entre 1 y . El argumento representa una diferencia de distancias, como se observa en la ecuación 4.5. A saber, es la diferencia de la distancia que hay entre la posición del sensor $\mathbf{t}_{g\,k}$ y el punto $\mathbf{p}$, menos la distancia medida por el sensor almacenada en R_k. La ecuación 4.7 calcula la posición $\mathbf{x}$ en el mapa de profundidad que le correspondería al punto $\mathbf{p}$ al ser transformado por la posición y orientación del sensor, y proyectado con la matriz de parámetros intrínsecos K. La ecuación 4.6 escala la distancia entre la posición del sensor $\mathbf{t}_{g\,k}$ y el punto $\mathbf{p}$, que se mide sobre el rayo que une esos puntos, a una medida de profundidad a lo largo del eje óptico del sensor.

La TSDF que se obtiene al aplicar esas ecuaciones es válida solamente para el punto de vista representado por la matriz $\mathrm{T}_{g\,k}$. Para unir las diferentes muestras, y formar una TSDF única, se utiliza un promedio ponderado de las diferentes muestras adquiridas. Al almacenar un peso $W_k(\mathbf{p})$ con cada valor de la función se puede obtener, incrementalmente y para cada punto $\mathbf{p}$, una función que se aproxime mejor a la superficie real como sigue:

$$F_k(\mathbf{p}) = \frac{W_{k-1}(\mathbf{p})F_{k-1}(\mathbf{p}) + W_{R_k}(\mathbf{p})F_{R_k}(\mathbf{p})}{W_{k-1}(\mathbf{p}) + W_{R_k}(\mathbf{p})} \tag{4.9}$$

$$W_k(\mathbf{p}) = W_{k-1}(\mathbf{p}) + W_{R_k}(\mathbf{p}) \tag{4.10}$$

En la práctica, según lo reportado en [25], basta con realizar un promedio simple, o sea, hacer $W_{R_k} = 1$, para obtener buenos resultados.

Esta parte del sistema está implementada en GPU. La función $\mathbf{S}_k(\mathbf{p})$ se almacena en un arreglo tridimensional en el que cada elemento representa espacio físico. Por ejemplo, cada voxel puede representar un espacio de $1\,mm^3$. La implementación procesa los *voxels* en rebanadas , de adelante hacia atrás, con una configuración de lanzamiento de hilos bidimensional. Como cada voxel puede convertirse a un punto equivalente $\mathbf{p}$, un hilo toma un voxel y realiza el proceso descrito en esta subsección, resumido a las ecuaciones 4.5 a 4.10. En [26] se puede encontrar una descripción de la implementación y, también, el proceso presentado en pseudocódigo.

4.1.3. Predicción de la super cie

En esta parte del sistema KinectFusion, se utiliza la técnica de trazado de rayos, en la posición y dirección de una cámara virtual representada por $\mathrm{T}_{g\,k}$, para encontrar *voxels* en los que el valor de F_k cambia, a lo largo del rayo, de positivo a negativo; es decir, se utiliza para encontrar el conjunto de nivel en el que $F_k = 0$. Los puntos encontrados se almacenan en $\mathbf{V}_k$ y para cada uno de estos puntos se calcula un vector normal $\mathbf{N}_k$ a la superficie representada en ese punto.

Para cada pixel $\mathbf{u}$, se traza un rayo con dirección $\mathrm{T}_{g\,k}\mathrm{K}^{-1}\mathbf{u}$ y se avanza en el volumen sobre este rayo en pasos menores a unidades, desde una distancia mínima, hasta encontrar un cruce en el que el valor de F_k cambie de positivo a negativo, indicando un cruce con cero. El avance sobre el rayo también termina cuando se encuentra un cruce de un valor negativo a uno positivo, cuando se alcanza una distancia máxima sobre el rayo, o cuando la celda correspondiente al punto del rayo sale de los límites del arreglo tridimensional.

En los puntos cercanos a la interfaz $F_k(\mathbf{p}) = 0$ se asume que el gradiente de la TSDF en $\mathbf{p}$ es ortogonal a ese conjunto de nivel y, en consecuencia, el vector normal que le corresponde al pixel $\mathbf{u}$ puede calcularse, utilizando derivadas numéricas, como:

$$\mathrm{R}_{g\,k}\mathbf{N}_{k-1} = \mathbf{N}_k^g = \quad F(\mathbf{p}), \quad F = \quad \frac{F}{x}, \frac{F}{y}, \frac{F}{z}^{\intercal} \tag{4.11}$$

En la computadora, como ya hemos mencionado, la función se almacena en un arreglo tridimensional. Cuando se encuentran dos puntos del rayo contenidos uno a distancia t, y en una celda de valor positivo; y otro a distancia $t+ \quad t$, y en una celda de valor negativo; se obtienen valores de la función interpolados de manera trilinear F_t y $F_{t+ \quad t}$. Para encontrar

con mayor precisión el punto de intersección, se toman esos valores y se calcula una distancia de intersección más precisa t utilizando:

$$t = t - \frac{tF_t}{F_{t+\ t} - F_t} \tag{4.12}$$

El punto $\mathbf{V}_k^g(\mathbf{u})$ es aquel, en el sistema de referencia global, que dista t unidades de la posición $\mathbf{t}_{g\,k}$ del sensor, medidas sobre el rayo correspondiente al pixel $\mathbf{u}$. En $\mathbf{V}_k(\mathbf{u})$ se almacena ese punto en el sistema de referencia k de la cámara, es decir, $\mathbf{V}_k(\mathbf{u})$=$\mathrm{T}_{g\,k}^{-1}\mathbf{V}_k^g(\mathbf{u})$. El vector almacenado en $\mathbf{N}_k(\mathbf{u})$ es el resultado de evaluar, numéricamente, el gradiente de la ecuación 4.11.

En [30] puede encontrarse una descripción más completa de la técnica de trazado de rayos y de la interpolación trilinear. En el siguiente capítulo se describe, también, el trazado de rayos.

4.1.4. Estimación de la posición y orientación

El propósito de esta etapa es obtener una transformación $\mathrm{T}_{g\,k}$ que convierta los puntos $\mathbf{V}_k(\mathbf{u})$ del cuadro actual k al sistema de referencia global g. Esta transformación, que representa la posición $\mathbf{t}_{g\,k}$, y la orientación (rotación) $\mathtt{R}_{g\,k}$ del sensor, tiene un papel importante en el sistema, como se muestra en el diagrama de la figura 4.1 y como se ha descrito en las subsecciones anteriores.

Para determinar esa transformación, el sistema utiliza un algoritmo ICP. Existen muchas variantes de este algoritmo, y pueden consultarse en [31], pero, en general, un algoritmo de este tipo asocia un punto de un conjunto A con otro punto destino de un conjunto B; encuentra una transformación que reduzca el error bajo alguna métrica entre los dos puntos (la distancia entre ellos, por ejemplo); transforma los puntos del conjunto A; e itera desde la asociación de puntos para obtener una mejor estimación de la transformación.

En este sistema, se busca la transformación $\mathrm{T}_{g\,k}$ que alinee[1] los puntos $\mathbf{V}_k$ obtenidos del mapa de profundidad actual (sección 4.1.1), con la superficie predicha representada por los puntos $\mathbf{V}_{k-1}^g$ (sección 4.1.3).

El primer paso es determinar correspondencias entre un punto del primer conjunto con otro del segundo. Con este fin, se asocia el punto $\mathbf{V}_k(\mathbf{u})$ con el punto $\mathbf{V}_{k-1}(\mathbf{u})$, donde $\mathbf{u}$ es el pixel que resulta de transformar, primero, el punto $\mathbf{V}_k(\mathbf{u})$ (en coordenadas homogéneas) al sistema de referencia del cuadro anterior mediante la transformación aproximada $\mathrm{T}_{k-1\,k}^z$ =

[1] La palabra que usualmente se utiliza es registrar, por traducción del inglés *register*, pero el autor no está convencido de utilizarla.

$\mathrm{T}^{-1}{}_{g\,k-1}\mathrm{T}^{z}_{g\,k}$ (donde $\mathrm{T}^{z}_{g\,k}$ es la estimación de la transformación en el cuadro actual y en la iteración z) y segundo, de proyectarlo con la matriz de parámetros intrínsecos K. Es decir, $\mathbf{u} = \mathrm{K}\mathrm{T}^{z}_{k-1\,k}\mathbf{V}_k(\mathbf{u})$. Los dos puntos $\mathbf{V}_k(\mathbf{u})$ y $\mathbf{V}_{k-1}(\mathbf{u})$ se asocian cuando se cumple que $\mathbf{V}_k(\mathbf{u})$ contiene una medición válida, cuando la distancia entre ellos, en el sistema de referencia global, es menor que un parámetro $_d$, y cuando los vectores normales en esos puntos forman cierto ángulo menor que un parámetro . O sea, cuando la función (**u**), que se muestra en la ecuación 4.13, es diferente de nulo.

$$(\mathbf{u}) = \text{nulo sii} \begin{cases} M_k(\mathbf{u}) = 1 & y \\ \mathrm{T}^{z}_{g\,k}\mathbf{V}_k(\mathbf{u}) - \mathbf{V}^{g}_{k-1}(\mathbf{u}) \;_2 \quad {}_d & y \\ \mathrm{R}^{z}_{g\,k}\mathbf{N}_k(\mathbf{u}) \;\; \mathbf{N}^{g}_{k-1}(\mathbf{u}) & \end{cases} \tag{4.13}$$

En la primera iteración $z = 0$ y la estimación de la transformación actual $\mathrm{T}^{z}_{g\,k}$ se inicializa con la matriz de transformación obtenida para el cuadro anterior $\mathrm{T}_{g\,k-1}$.

El siguiente paso es determinar la transformación $\mathrm{T}_{g\,k}$ que minimiza la suma del cuadrado de la distancia de cada punto $\mathbf{V}_k(\mathbf{u})$ al plano que contiene su correspondiente $\mathbf{V}^{g}_{k-1}(\mathbf{u})$ y que tiene por normal el vector $\mathbf{N}^{g}_{k-1}(\mathbf{u})$. Con una fórmula, se busca $\mathrm{T}_{g\,k}$ que minimice:

$$\underset{\substack{\mathbf{u}\ \mathcal{U} \\ (\mathbf{u})=nulo}}{} \mathrm{T}_{g\,k}\mathbf{V}_k(\mathbf{u}) - \mathbf{V}^{g}_{k-1}(\mathbf{u}) \quad \mathbf{N}^{g}_{k-1}(\mathbf{u}) \;^2 \tag{4.14}$$

La transformación $\mathrm{T}_{g\,k}$, o más bien, su aproximación $\mathrm{T}^{z}_{g\,k}$, se obtiene con una linearización de la ecuación 4.14 y por iteraciones. Para cada iteración z, se obtiene $\mathrm{T}^{z}_{g\,k} = \mathrm{T}^{z}_{inc}\mathrm{T}^{z-1}_{g\,k}$ a partir de la iteración anterior $\mathrm{T}^{z-1}_{g\,k}$ y de una transformación de incremento T^{z}_{inc}.

La matriz de incremento T^{z}_{inc} se forma por una matriz de rotaciones[2] simplificada utilizando la aproximación para ángulos pequeños, en la que si 0, sin y cos 1, y por un vector de traslación. Es decir,

$$\mathrm{T}^{z}_{inc} = \mathrm{R}^{z} \;\; \mathbf{t}^{z} \;\; = \begin{pmatrix} 1 & \alpha & -\gamma & t_x \\ -\alpha & 1 & \beta & t_y \\ \gamma & -\beta & 1 & t_z \end{pmatrix}. \tag{4.15}$$

Los parámetros de esa matriz se pueden escribir como un vector:

$$\mathbf{x} = (\beta, \gamma, \alpha, t_x, t_y, t_z) \in \mathbb{R}^6. \tag{4.16}$$

[2]Por ejemplo, la que se muestra en la ecuación 5.1.

Con este vector y con los puntos actuales transformados con la estimación de la iteración anterior $\mathbf{V}_k^g(\mathbf{u}) = \mathrm{T}_{g\,k}^{z\,-1}\mathbf{V}_k(\mathbf{u})$, podemos escribir:

$$\begin{aligned}
\mathrm{T}_{g\,k}^{z}\mathbf{V}_k(\mathbf{u}) &= \mathrm{T}_{inc}^{z}\mathrm{T}_{g\,k}^{z\,-1}\mathbf{V}_k(\mathbf{u}) \\
&= \mathrm{T}_{inc}^{z}\mathbf{V}_k^g(\mathbf{u}) \\
&= \mathtt{R}^z\mathbf{V}_k^g(\mathbf{u}) + \mathbf{t}^z \\
&= \begin{pmatrix} 1 & \alpha & -\gamma \\ -\alpha & 1 & \beta \\ \gamma & -\beta & 1 \end{pmatrix} \begin{pmatrix} \tilde{v}_x(\mathbf{u}) \\ \tilde{v}_y(\mathbf{u}) \\ \tilde{v}_z(\mathbf{u}) \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \\
&= \begin{pmatrix} \tilde{v}_x(\mathbf{u}) + \alpha\tilde{v}_y(\mathbf{u}) - \gamma\tilde{v}_z(\mathbf{u}) + t_x \\ -\alpha\tilde{v}_x(\mathbf{u}) + \tilde{v}_y(\mathbf{u}) + \beta\tilde{v}_z(\mathbf{u}) + t_y \\ \gamma\tilde{v}_x(\mathbf{u}) - \beta\tilde{v}_y(\mathbf{u}) + \tilde{v}_z(\mathbf{u}) + t_z \end{pmatrix} \\
&= \begin{pmatrix} \alpha\tilde{v}_y(\mathbf{u}) - \gamma\tilde{v}_z(\mathbf{u}) + t_x \\ -\alpha\tilde{v}_x(\mathbf{u}) + \beta\tilde{v}_z(\mathbf{u}) + t_y \\ \gamma\tilde{v}_x(\mathbf{u}) - \beta\tilde{v}_y(\mathbf{u}) + t_z \end{pmatrix} + \begin{pmatrix} \tilde{v}_x(\mathbf{u}) \\ \tilde{v}_y(\mathbf{u}) \\ \tilde{v}_z(\mathbf{u}) \end{pmatrix}
\end{aligned}$$

$$\begin{aligned}
\mathrm{T}_{g\,k}^{z}\mathbf{V}_k(\mathbf{u}) &= \begin{pmatrix} \alpha\tilde{v}_y(\mathbf{u}) - \gamma\tilde{v}_z(\mathbf{u}) + t_x \\ -\alpha\tilde{v}_x(\mathbf{u}) + \beta\tilde{v}_z(\mathbf{u}) + t_y \\ \gamma\tilde{v}_x(\mathbf{u}) - \beta\tilde{v}_y(\mathbf{u}) + t_z \end{pmatrix} + \begin{pmatrix} \tilde{v}_x(\mathbf{u}) \\ \tilde{v}_y(\mathbf{u}) \\ \tilde{v}_z(\mathbf{u}) \end{pmatrix} \\
&= \begin{pmatrix} 0 & -\tilde{v}_z(\mathbf{u}) & \tilde{v}_y(\mathbf{u}) & 1 & 0 & 0 \\ \tilde{v}_z(\mathbf{u}) & 0 & -\tilde{v}_x(\mathbf{u}) & 0 & 1 & 0 \\ -\tilde{v}_y(\mathbf{u}) & \tilde{v}_x(\mathbf{u}) & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \beta \\ \gamma \\ \alpha \\ t_x \\ t_y \\ t_z \end{pmatrix} + \begin{pmatrix} \tilde{v}_x(\mathbf{u}) \\ \tilde{v}_y(\mathbf{u}) \\ \tilde{v}_z(\mathbf{u}) \end{pmatrix} \qquad (4.17) \\
&= \mathbf{G}(\mathbf{u})\mathbf{x} + \mathbf{V}_k^g(\mathbf{u})
\end{aligned}$$

La matriz $\mathbf{G}(\mathbf{u})$, de 3×6, se forma con la matriz antisimétrica del vector $\mathbf{V}_k^g(\mathbf{u})$ y con la matriz identidad de 3×3. O sea,

$$\mathbf{G}(\mathbf{u}) = \quad \mathbf{V}_k^g(\mathbf{u}) \quad \mathbf{I}_{3\ 3} \quad . \qquad (4.18)$$

Con el resultado de la ecuación 4.17 podemos reescribir la expresión 4.14, cambiando el producto punto por una multiplicación de matrices:

$$\underset{\substack{\mathbf{u}\ \mathcal{U} \\ (\mathbf{u})=nulo}}{} \mathbf{N}_{k-1}^{g}(\mathbf{u})^{\intercal} \quad \mathbf{G}(\mathbf{u})\mathbf{x} + \mathbf{V}_{k}^{g}(\mathbf{u}) - \mathbf{V}_{k-1}^{g}(\mathbf{u}) \quad ^{2} \tag{4.19}$$

Al distribuir la multiplicación tenemos:

$$\underset{\substack{\mathbf{u}\ \mathcal{U} \\ (\mathbf{u})=nulo}}{} \mathbf{N}_{k-1}^{g}(\mathbf{u})^{\intercal}\mathbf{G}(\mathbf{u})\mathbf{x} + \mathbf{N}_{k-1}^{g}(\mathbf{u})^{\intercal} \quad \mathbf{V}_{k}^{g}(\mathbf{u}) - \mathbf{V}_{k-1}^{g}(\mathbf{u}) \quad ^{2} \tag{4.20}$$

Y al definir las matrices $\mathbf{A}$ y $\mathbf{b}$ de esta manera:

$$\mathbf{A} = \mathbf{N}_{k-1}^{g}(\mathbf{u})^{\intercal}\mathbf{G}(\mathbf{u}) \tag{4.21}$$

$$\mathbf{b} = \mathbf{N}_{k-1}^{g}(\mathbf{u})^{\intercal} \quad \mathbf{V}_{k-1}^{g}(\mathbf{u}) - \mathbf{V}_{k}^{g}(\mathbf{u}) \quad , \tag{4.22}$$

podemos escribir la ecuación 4.20 como:

$$\underset{\substack{\mathbf{u}\ \mathcal{U} \\ (\mathbf{u})=nulo}}{} \mathbf{A}\mathbf{x} - \mathbf{b} \quad ^{2} . \tag{4.23}$$

Con esto, el problema se redujo a encontrar el vector $\mathbf{x}$ que minimiza la suma de la ecuación 4.23. Y es el problema de mínimos cuadrados.

Para encontrar el vector $\mathbf{x}$ que minimiza la suma, se requiere derivar la expresión respecto de $\mathbf{x}$ e igualar el resultado a cero. Para eso, primero desarrollamos como sigue:

$$\begin{aligned}
\underset{\substack{\mathbf{u}\ \mathcal{U} \\ (\mathbf{u})=nulo}}{} \mathbf{A}\mathbf{x} - \mathbf{b} \quad ^{2} &= \underset{\substack{\mathbf{u}\ \mathcal{U} \\ (\mathbf{u})=nulo}}{} (\mathbf{A}\mathbf{x} - \mathbf{b})^{\intercal} (\mathbf{A}\mathbf{x} - \mathbf{b}) \\
&= \underset{\substack{\mathbf{u}\ \mathcal{U} \\ (\mathbf{u})=nulo}}{} ((\mathbf{A}\mathbf{x})^{\intercal} - \mathbf{b}^{\intercal}) (\mathbf{A}\mathbf{x} - \mathbf{b}) \\
&= \underset{\substack{\mathbf{u}\ \mathcal{U} \\ (\mathbf{u})=nulo}}{} (\mathbf{A}\mathbf{x})^{\intercal} (\mathbf{A}\mathbf{x}) - (\mathbf{A}\mathbf{x})^{\intercal} \mathbf{b} - \mathbf{b}^{\intercal} (\mathbf{A}\mathbf{x}) + \mathbf{b}^{\intercal}\mathbf{b} \\
&= \underset{\substack{\mathbf{u}\ \mathcal{U} \\ (\mathbf{u})=nulo}}{} \mathbf{x}^{\intercal}\mathbf{A}^{\intercal}\mathbf{A}\mathbf{x} - \mathbf{x}^{\intercal}\mathbf{A}^{\intercal}\mathbf{b} - \mathbf{b}^{\intercal}\mathbf{A}\mathbf{x} + \mathbf{b}^{\intercal}\mathbf{b} \\
&= \underset{\substack{\mathbf{u}\ \mathcal{U} \\ (\mathbf{u})=nulo}}{} \mathbf{x}^{\intercal}\mathbf{A}^{\intercal}\mathbf{A}\mathbf{x} - 2\mathbf{b}^{\intercal}\mathbf{A}\mathbf{x} + \mathbf{b}^{\intercal}\mathbf{b}
\end{aligned}$$

Y derivamos respecto a $\mathbf{x}$:

$$\frac{(\mathbf{x}^\intercal \mathbf{A}^\intercal \mathbf{A}\mathbf{x} - 2\mathbf{b}^\intercal \mathbf{A}\mathbf{x} + \mathbf{b}^\intercal \mathbf{b})}{\mathbf{x}} = 0$$
$$\mathbf{u}\ \mathcal{U}\ (\mathbf{u})=nulo$$

$$2\mathbf{A}^\intercal \mathbf{A}\mathbf{x} - 2\mathbf{A}^\intercal \mathbf{b} = 0$$
$$\mathbf{u}\ \mathcal{U}\ (\mathbf{u})=nulo$$

$$\mathbf{A}^\intercal \mathbf{A}\mathbf{x} = \mathbf{A}^\intercal \mathbf{b} \tag{4.24}$$
$$\mathbf{u}\ \mathcal{U}\ (\mathbf{u})=nulo \qquad \mathbf{u}\ \mathcal{U}\ (\mathbf{u})=nulo$$

En cada iteración, y para cada par de puntos asociados, $\mathbf{V}_k(\mathbf{u})$ y $\mathbf{V}^g_{k-1}(\mathbf{u})$ se forma un sistema lineal de 6×6. Las operaciones $\mathbf{A}^\intercal \mathbf{A}$ y $\mathbf{A}^\intercal \mathbf{b}$ de cada pareja de puntos asociados se realizan en GPU (haciendo uso del hecho que la primera matriz, $\mathbf{A}^\intercal \mathbf{A}$, es simétrica, para ahorrar operaciones). La suma de todas estas matrices y vectores resultantes también se realiza en GPU con una reducción similar a la del ejemplo de la sección 2.6. El sistema lineal final se resuelve en CPU con el método de descomposición de Cholesky, y el vector resultante se convierte en una matriz de rotación y traslación.

Para cada cuadro, se itera sobre los niveles 3,2 y 1 de la pirámide descrita en la sección 4.1.1 por $4, 5$ y 10 iteraciones, respectivamente. Al final de esas iteraciones se asigna la posición y orientación actual $\mathrm{T}_{g\,k}$ $\mathrm{T}^z_{g\,k}$. Antes de asignar la estimación actual, se verifica que los parámetros del vector $\mathbf{x}$ no hayan excedido un límite y no hayan roto la suposición de los ángulos pequeños con que se obtuvo la forma lineal. Se verifica, también, que hayan existido suficientes asociaciones para el sistema de la ecuación 4.24. Cuando alguna de esas pruebas falla, se asigna la última posición correcta conocida.

4.2. Trabajos recientes

Después de conocer el trabajo reescrito en la sección anterior, surgen algunas preguntas. Por ejemplo, habrá otras otras maneras de almacenar la TSDF? Qué procesamiento puede hacerse sobre el mapa de profundidad para eliminar la falta de información y el ruido del sensor? Existen otras formas de estimar la posición y orientación del sensor además de algoritmos ICP? Cómo trabajar con escenas dinámicas? Los trabajos recientes sobre este tema se enfocan en algunas de estas preguntas y se presentan, brevísimamente, a continuación.

4.2.1. Almacenamiento de la TSDF

A manera de ejemplo de dos enfoques diferentes, está [32], que utiliza un arreglo tridimensional con celdas jerárquicas, de manera que cada *voxel* se subdivide en otros solamente si la TSDF contiene valores no truncados; y también está [33], que almacena únicamente los *voxels* que contienen valores de la TSDF y almacenados en una tabla *hash*.

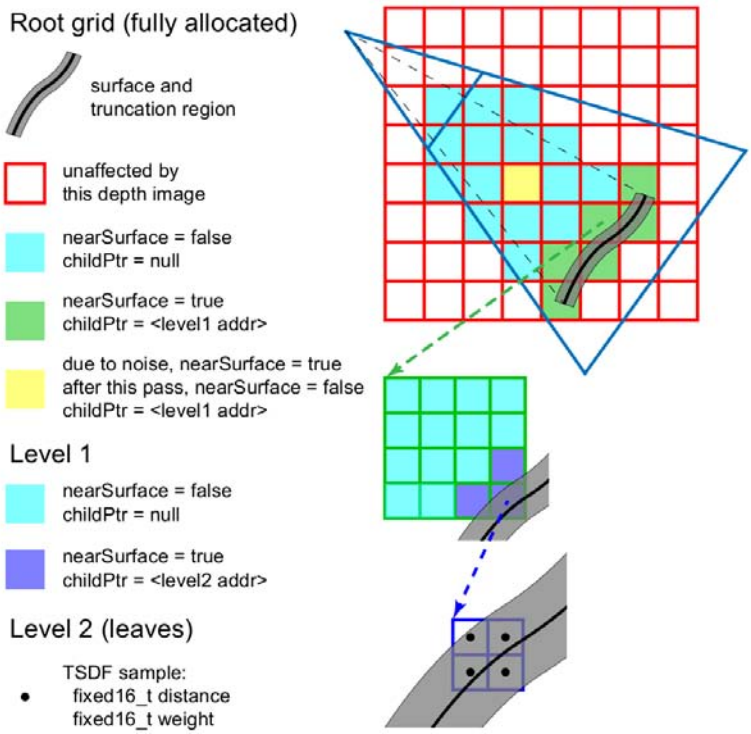


FIGURA 4.4: La estructura jerárquica utilizada en [32].

La estructura que se utiliza en el primer trabajo se muestra, como fue presentada en [32], en la figura 4.4. Consta de tres niveles, el nivel cero, la raíz, se muestra en rojo y corresponde a todas las celdas (*voxels*) que están almacenadas en memoria y que representan todo el espacio observable.

Estas celdas pueden representar espacio observado vacío, en cian, o espacio observado cercano a una superficie, en verde. Las celdas en verde contienen, además de la bandera `nearSurface`, que representa que esta celda está cerca de una superficie, un apuntador al siguiente nivel.

En el primer nivel se tiene algo similar, cada celda almacena una bandera que se activa cuando cuando la celda contiene parte de la superficie, en negro, o de la región de truncamiento, en gris; y un apuntador al siguiente nivel en ese caso. El segundo nivel almacena el peso W_k y el valor F_k de la TSDF.

El sistema funciona, esencialmente, como el descrito en la sección anterior, pero, necesariamente, con diferencias en cómo se accede a cada *voxel* de la estructura de datos cuando se actualiza el modelo y cuando se realiza el trazado de rayos. El propósito de [32] es tener una estructura de datos eficiente para esas dos etapas y que permita representar espacios grandes con detalle, y en ese artículo se describe cómo fue implementada en GPU.

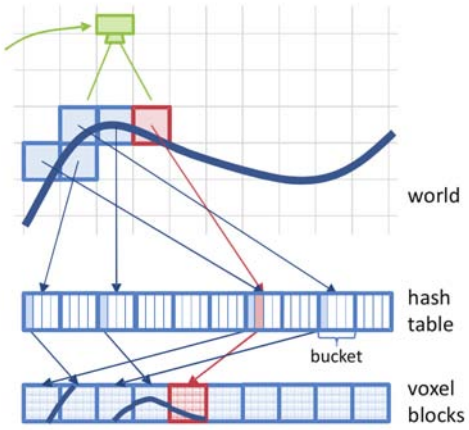


FIGURA 4.5: La tabla *hash* utilizada en [33].

El trabajo de [33] subdivide el espacio en celdas también, pero almacena solamente las que están cerca de una superficie, y lo hace en una tabla *hash*, como se muestra en la figura 4.5. Cada *voxel* almacena el valor F_k de la TSDF, el peso W_k que se le asigna a ese valor, y el color; y cada *voxel* se almacena, con otros, en bloques que contienen 8^3 de ellos (*voxel blocks*).

El *voxel* con coordenadas (x, y, z) se asocia con el lugar que le corresponde en la tabla $H(x, y, z)$ mediante la función *hash* H. Más de un *voxel* puede estar asociado con una misma ubicación de la tabla, y para resolver estas colisiones, cada elemento de la tabla se subdivide en cubetas (*buckets*). Cada elemento de esas cubetas almacena la posición (x, y, z) del *voxel*, un apuntador al bloque en que se encuentra, y un salto (*offset*) a otra cubeta (que se utiliza cuando se desborda una de esas cubetas).

En este artículo se describen todas las operaciones necesarias para mantener un flujo parecido al que estudiamos en la sección anterior. Por ejemplo, la inserción de *voxels* a la tabla, la eliminación de ellos, la actualización del modelo de la superficie, la predicción de la superficie, etc.

Ambos artículos buscan tener una estructura de almacenamiento de la TSDF que aproveche mejor la dispersión de celdas que contienen información cercana a la superficie. Dado que en un arreglo fijo tridimensional (como el del sistema KinectFusion o el que se presenta

en el siguiente capítulo) la mayoría de las celdas representan espacio vacío (pero ocupan espacio en la memoria), capturar espacios más grandes crece cúbicamente en memoria.

4.2.2. Procesamiento del mapa de profundidad

Como ejemplos de trabajos dedicados a mejorar los mapas de profundidad está [34], que expone un método para la eliminación de huecos (ausencia de mediciones) en un mapa de profundidad; y [35] que hace un análisis de los errores introducidos en el mapa por emborronamiento debido al movimiento (*motion blur*).

En el capítulo 3 vimos, someramente, algunas causas de la falta de información en los mapas de profundidad calculados mediante luz estructurada. En el primer ejemplo, [34], se propone un método para disminuir esta falta de información y el flujo se muestra en la figura 4.6.

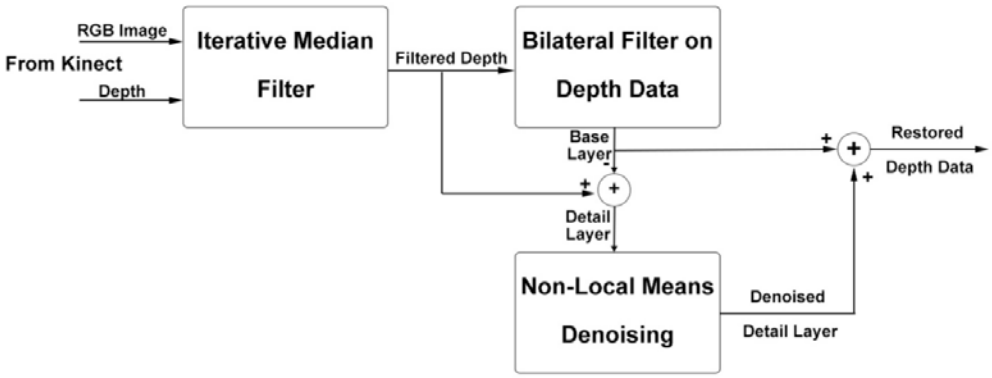


FIGURA 4.6: Remoción de huecos en un mapa de profundidad [34].

El primer bloque, el filtro de mediana iterativo (*Iterative Median Filter)*, asocia pixeles sin información de profundidad con otros de su vecindad según su similitud de color para determinar un valor de profundidad. Este proceso se repite hasta cumplir con un número máximo de huecos permitidos.

El segundo bloque es como el filtro bilateral que estudiamos en la sección 4.1.1.

La resta del mapa filtrado por el primer bloque menos el mapa filtrado en el segundo proporciona detalles sobre los bordes, y para disminuir el ruido en ellos, se utilizan métodos que no trabajan sobre una vecindad, sino en todos los pixeles similares del mapa de profundidad (métodos no locales).

La imagen final es la suma de la imagen producida por el segundo bloque más la imagen producida por el tercero.

El trabajo de [35] busca eliminar el error en los mapas de profundidad por objetos que se mueven. Su sistema analiza una secuencia de imágenes, encuentra la zona en que el objeto se mueve, mejora esa zona, y con la información de diferentes cuadros de profundidad en el tiempo, genera la información faltante.

4.2.3. Estimación de la posición y orientación

Vimos, en el análisis del sistema KinectFusion, que un elemento importante es la transformación $T_{g\,k}$, que convierte los puntos a un mismo sistema de referencia, y que en ese sistema se obtiene con un algoritmo ICP. Existen, sin embargo, otras maneras.

Como primer ejemplo, en [36] realizan una estimación inicial de la transformación utilizando descriptores de imágenes y de conjuntos de puntos para generar correspondencias y estimar la transformación utilizando RANSAC (*Random Sample Consensus*)[3], esta estimación se refina con ICP y se integra con un flujo como el estudiado.

Otro ejemplo es [38]. En ese trabajo se sigue una metodología distinta. Utilizan descriptores de la información de color y geométrica, así como técnicas de construcción a partir de movimiento (*StM, Structure from Motion*) que ocupan la secuencia de imágenes para construir mejores mapas de profundidad. Con una variación de RANSAC que utiliza los descriptores para estimar la posición, dan una solución al problema llamado *drifting*, que en español podría traducirse como desorientación , puesto que se trata de la pérdida de la orientación en escenas con fragmentos que no proporcionan información del giro del sensor, como una pared plana y continua. También dan una solución al problema de *loop-closure*, que consiste en determinar si el sensor ya ha estado en una posición y hacer las correcciones necesarias al modelo para un mejor alineado.

Estos dos enfoques no apuntan a construir representaciones en tiempo real, pero presentan resultados interesantes y relacionados al tema. Se recomienda un análisis más profundo del funcionamiento del segundo.

4.2.4. Escenas dinámicas

Los cambios en la escena (gente u objetos que se mueven) pueden llevar a una pobre estimación de la matriz de transformación $T_{g\,k}$. Algunos trabajos, como [39], [40] y [41] se enfocan en reconstruir escenas con cambios.

El trabajo de [39] consta de cuatro etapas. En la primera, se procesa un mapa de profundidad entrante y se convierte a puntos con la transformación estimada hasta ese

[3] En otros trabajos de tesis de generaciones anteriores de este posgrado ya se ha hablado de descriptores de imágenes y del algoritmo RANSAC para estimar parámetros de un modelo matemático. Por ejemplo, en [37].

momento. La segunda actualiza el modelo, pero este sistema no utiliza una estructura de datos, sino que almacena la lista de puntos tridimensionales y, para cada uno de ellos, un vector normal, un radio (que se utiliza para generar el equivalente a la superficie de la sección 4.1.3), y una serie de atributos que marcan a un punto como inestable o estable (o qué tanto aparece en una escena). La tercera etapa, estima la posición con un algoritmo ICP. La cuarta etapa se encarga de la actualización de los atributos de estabilidad, según la frecuencia de observación de ese punto.

En [40] se busca generar una representación de un objeto, como una cara, una mano, un torso, o alguna cosa, que cambie de forma en el tiempo. Tiene dos fases. En la primera, se genera, utilizando el sistema de [33], un modelo del objeto que se usa en la segunda fase. El modelo se convierte en una malla de triángulos y, en la segunda fase, se transforma en tiempo real según la información que proporciona el sensor.

El reciente trabajo de [41] apunta a extender KinectFusion a escenas enteramente dinámicas, sin necesidad de un patrón sobre el cual se realicen las deformaciones.

Cap´tulo 5

Sistema ejemplo

El sistema que se describe en este capítulo surgió como una primera solución al problema y para ganar entendimiento de las dificultades que emergen. Es, por tanto, distante a los del estado actual descrito en el capítulo anterior. Nuestro sistema ejemplo está constituido por dos módulos que corresponden a los dos propósitos del trabajo de esta tesis, a saber, construir una representación del entorno de un robot (fig. 5.1) con un sensor RGB-D, y simular lo que el sensor capturaría en posiciones y orientaciones arbitrarias. El primer módulo convierte los mapas de profundidad a un sistema de coordenadas globales, utilizando la posición y orientación del sensor dadas por los sistemas de medición del robot, y actualiza el modelo del ambiente. El segundo módulo produce un mapa de profundidad de un sensor en una posición y orientación arbitrarias utilizando el modelo producido por el primer módulo.

El sistema funciona suponiendo que el robot puede proporcionar, a través de sus sistemas de movimiento y para cada cuadro, la información de los seis grados de libertad del sensor RGB-D que está montado en él. Es decir, que puede proporcionar la posición (x, y, z) y orientación (α, β, γ) del sensor RGB-D, o un conjunto de parámetros que permita calcular los seis mencionados[1]. La matriz K de parámetros intrínsecos —como la descrita en el capítulo tres— del sensor RGB-D es conocida también.

Las partes paralelizables están implementadas en GPU mediante CUDA C y las secuenciales en C++ y C.

[1]Por ejemplo, la posición (x, y) del robot en el plano de su movimiento, el ángulo θ de rotación del robot alrededor de un eje que corta la estructura del robot y es perpendicular al plano de su movimiento, y los ángulos del movimiento de la cabeza α, que mide el giro del sensor a la izquierda y a la derecha , y γ, que mide el giro del sensor hacia arriba y hacia abajo .

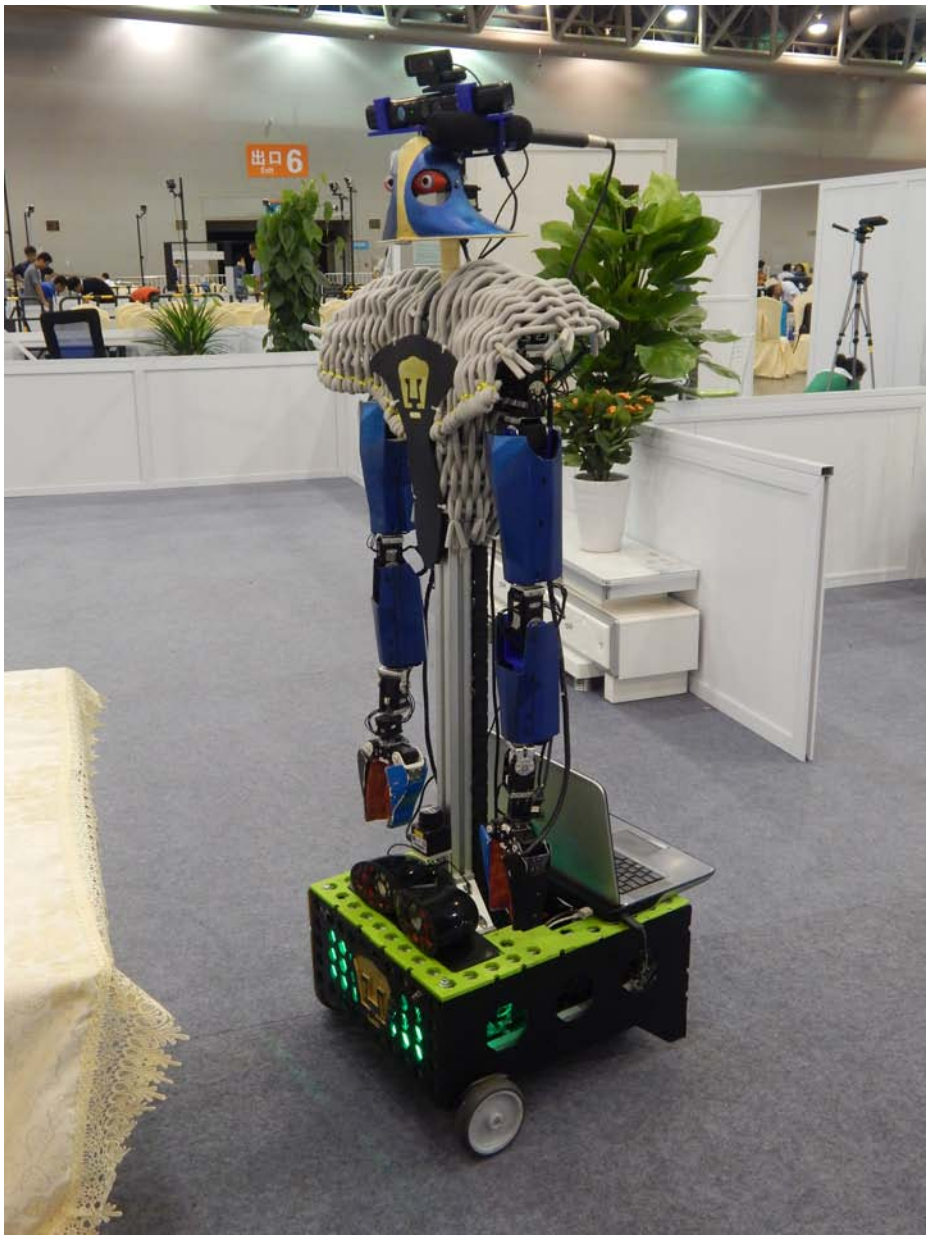


Figura 5.1: El robot Justina del laboratorio de Biorobótica de la Facultad de Ingeniería. En la parte superior puede observarse el sensor Kinect.

5.1. Construcción del ambiente

El propósito de este módulo es tener una representación en la computadora del ambiente que rodea al robot. La representación que hemos escogido es un arreglo tridimensional $S(\mathbf{v})$ en donde cada elemento $\mathbf{v} = (i_x, i_y, i_z)$ (*voxel*), con $i_x, i_y, i_z \in \mathbb{Z}^+$, representa $1\,cm^3$ del espacio físico que rodea al robot. Cada una de estas celdas es, en la memoria de la computadora, un *byte* que toma un valor positivo cuando el espacio representado por el voxel $\mathbf{v}$ está ocupado y un cero cuando el voxel representa espacio libre. En otras palabras, el valor $S(\mathbf{v})$ es positivo cuando el sensor midió una distancia a una superficie y cero cuando no.

Este modelo se actualiza con las medidas de profundidad que proporciona el sensor RGB-D. El sensor captura información desde diferentes orientaciones y posiciones, por lo que hay que convertir, primero, las profundidades a un sistema de coordenadas de la cámara y convertir, después, esos puntos del sistema de la cámara a un sistema de referencia global. El primer paso se logra con la matriz K de parámetros intrínsecos del sensor y con las ecuaciones 3.7, como describimos en el capítulo tres. El segundo paso utiliza una matriz de transformación $\mathtt{T}_g$ formada con los datos de posición y orientación del sensor, utilizando la ecuación 3.8. Aunque se puede escoger cualquier sistema de referencia global arbitrario, las mediciones del robot son fáciles de entender con el que a continuación se describe.

Escogimos un sistema cartesiano tridimensional que sigue la regla de la mano derecha y que tiene ejes X, Y y Z. El plano XY es horizontal (paralelo al piso) y contiene el origen, que corresponde con el primer lugar desde el que el sensor capturó información. El eje Z es vertical, perpendicular al plano XY, pasa por el origen y es positivo hacia arriba. El eje Y está contenido en el plano XY y es positivo en la dirección del desplazamiento hacia adelante del robot. El eje X es perpendicular al Y y positivo a la derecha. Los ángulos de giro α, β y γ corresponden al giro alrededor del eje Z, Y y X, respectivamente, y son positivos cuando el giro ocurre en el sentido antihorario (contrario a las manecillas del reloj). La ventaja de tener el sensor montado en un robot humanoide es que estos ángulos pueden describirse fácilmente imaginando que uno es el robot. Así, el ángulo de giro α, al rededor del eje Z, corresponde a voltear la cabeza a la derecha y a la izquierda, y es positivo en este último caso. El ángulo β, que mide el giro alrededor del eje Y, equivale a llevar la oreja izquierda al hombro izquierdo y la oreja derecha al hombro derecho, positivo en el último caso. Finalmente, el ángulo de giro γ, alrededor del eje X, equivale a voltear hacia arriba y hacia abajo, siendo positivo en aquel caso. La figura 5.2 muestra los ejes del sistema de referencia y los ángulos de giro.

Para convertir los puntos del sistema de la cámara al sistema global elegido, la ecuación 3.8 utiliza la posición $(x_s, y_s, z_s)^\intercal$ del sensor en el sistema de referencia global como el vector

de traslación, y los ángulos α, β y γ para formar los términos de las rotaciones en torno a los ejes Z, Y y X. Entonces, la transformación necesaria es:

$$
\begin{aligned}
\mathtt{T}_g &= \mathfrak{T}_{xyz}\mathtt{R}_z\mathtt{R}_y\mathtt{R}_x \\
&= \begin{pmatrix} 1 & 0 & 0 & x_s \\ 0 & 1 & 0 & y_s \\ 0 & 0 & 1 & z_s \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} \cos\alpha\cos\beta & -\sin\alpha\cos\gamma + \cos\alpha\sin\beta\sin\gamma & \sin\alpha\sin\gamma + \cos\alpha\sin\beta\cos\gamma & x_s \\ \sin\alpha\cos\beta & \cos\alpha\cos\gamma + \sin\alpha\sin\beta\sin\gamma & -\cos\alpha\sin\gamma + \sin\alpha\sin\beta\sin\gamma & y_s \\ -\sin\beta & \cos\beta\sin\gamma & \cos\beta\cos\gamma & z_s \\ 0 & 0 & 0 & 1 \end{pmatrix}
\end{aligned} \tag{5.1}
$$

Con todo lo anterior podemos describir con más facilidad el procedimiento para convertir las profundidades de un mapa D a una lista de puntos P en un sistema de coordenadas global g. Para cada pixel $\mathbf{u} = (u, v)$ del mapa de profundidad (donde u y v son enteros, se miden desde la esquina superior izquierda de la imagen y son positivos, respectivamente, hacia la derecha y hacia abajo) se obtienen las coordenadas de cámara con las ecuaciones 3.7 y utilizando la matriz K. O sea, obtenemos

$$
\begin{pmatrix} x_c \\ y_c \\ z_c \end{pmatrix} = \begin{pmatrix} \dfrac{(u - c_x)D(\mathbf{u})}{f_x} \\ \dfrac{(v - c_y)D(\mathbf{u})}{f_y} \\ D(\mathbf{u}) \end{pmatrix}. \tag{5.2}
$$

Estas coordenadas están referidas respecto a un sistema en el que el eje X es horizontal y positivo a la derecha, el eje Y vertical y positivo hacia abajo, y el eje Z indica la profundidad. Este sistema es ligeramente distino al que nosotros elegimos, en el que el eje Y es el que marca la profundidad y Z es vertical pero positivo hacia arriba. Entonces, solamente hay que hacer:

$$
\begin{pmatrix} x'_c \\ y'_c \\ z'_c \end{pmatrix} = \begin{pmatrix} x_c \\ z_c \\ -y_c \end{pmatrix} \tag{5.3}
$$

para obtener las coordenadas en el sistema que elegimos.

Con estas coordenadas de cámara $(x'_c, y'_c, z'_c)^\intercal$ en el sistema que nosotros elegimos, podemos obtener las coordenadas globales (x_g, y_g, z_g) multiplicando la matriz de transformación

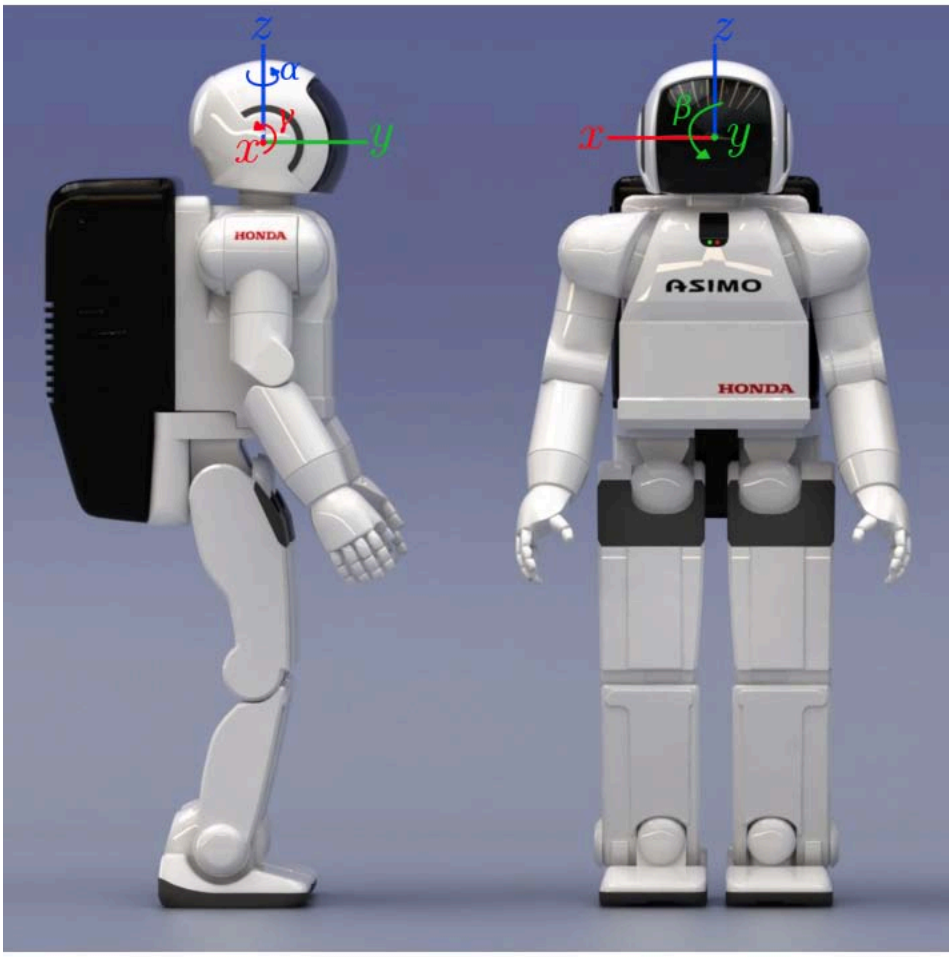


FIGURA 5.2: Los ejes X, Y y Z se muestran en rojo, verde y azul, respectivamente, sobre el robot Asimo de la compañía Honda.[42]

$\mathtt{T}_g$ que describimos por aquellas coordenadas, como en la ecuación 3.8. Es decir,

$$\begin{pmatrix} x_g \\ y_g \\ z_g \end{pmatrix} = \begin{pmatrix} \cos\alpha\cos\beta & -\sin\alpha\cos\gamma + \cos\alpha\sin\beta\sin\gamma & \sin\alpha\sin\gamma + \cos\alpha\sin\beta\cos\gamma & x_s \\ \sin\alpha\cos\beta & \cos\alpha\cos\gamma + \sin\alpha\sin\beta\sin\gamma & -\cos\alpha\sin\gamma + \sin\alpha\sin\beta\sin\gamma & y_s \\ -\sin\beta & \cos\beta\sin\gamma & \cos\beta\cos\gamma & z_s \end{pmatrix} \begin{pmatrix} x'_c \\ y'_c \\ z'_c \\ 1 \end{pmatrix}. \quad (5.4)$$

Los enteros de nuestras coordenadas globales representan, hasta ahora, milímetros, puesto que esta es la unidad que proporciona la medición $D(\mathbf{u})$. Como nosotros hemos escogido por unidad el centímetro, solamente hace falta escalar cada coordenada por 0.1 para obtener los puntos V como serán almacenados en el arreglo tridimensional descrito en el primer párrafo de esta sección.

A manera de resumen del proceso, se presenta el pseudocódigo 1.

Pseudocódigo 1: Mapa de profundidad al sistema de referencia global

input : El mapa de profundidad D de 640×480 pixeles, la matriz de parámetros intrínsecos K del sensor, la matriz de transformación T (con elementos $\mathtt{T}_{i\,j}$ donde i es el renglón y j la columna) que representa la posición y orientación del sensor en el sistema global. La escala s necesaria para que la parte entera de las coordenadas corresponda a la unidad de longitud elegida.

output: Una lista de puntos P en el sistema de referencia global.

```
1 foreach pixel u = (u, v) in parallel do
2     x_c   (u − c_x)   D[u, v]   f_x
3     y_c   D[u, v]
4     z_c   −1   (v − c_y)   D[u, v]   f_y;
5     (x_g, y_g, z_g)^⊺   T   (x_c, y_c, z_c)^⊺
6     P[u, v]   (s   x_g, s   y_g, s   z_g)
```

Hasta ahora hemos obtenido, a partir de cada medición del mapa de profundidad D, una lista de puntos P, pero no hemos dicho nada sobre cómo estos puntos se almacenan en la estructura elegida. Aunque la parte entera de los puntos ya representa centímetros, aún hay coordenadas con elementos negativos y que no pueden utilizarse como índices del arreglo tridimensional, dado que estos deben ser enteros positivos. Esto se resuelve fácilmente trasladando el origen del sistema de referencia al centro del arreglo tridimensional.

Con las dimensiones V_x, V_y y V_z del ancho, profundidad y altura del arreglo podemos obtener los índices $(i_x, i_y, i_z)^\intercal$ que le corresponden al punto $(x_g, y_g, z_g)^\intercal$ de P dentro del arreglo con

$$\begin{pmatrix} i_x \\ i_y \\ i_z \end{pmatrix} = \begin{pmatrix} x_g + \frac{V_x}{2} \\ y_g + \frac{V_y}{2} \\ z_g + \frac{V_z}{2} \end{pmatrix}, \tag{5.5}$$

y se descartan los puntos cuyos índices excedan las dimensiones del arreglo o se mantengan negativos.

Estos índices permiten actualizar el modelo recorriendo la lista de puntos P y escribiendo un valor 255 en la celda que representa a cada punto.

Todo el procedimiento se realiza para cada cuadro capturado por el sensor de profundidad. Al final del procesamiento tendremos una representación tridimensional en la que cada *voxel* $V(i_x, i_y, i_z)$ que representa un espacio físico de $1\,cm^3$ puede estar ocupado o desocupado. O sea,

$$V(i_x, i_y, i_z) = \begin{cases} 255 & \text{si el } \textit{voxel} \text{ representa espacio ocupado} \\ 0 & \text{si el } \textit{voxel} \text{ representa espacio desocupado} \end{cases} \tag{5.6}$$

Aunque nosotros hemos elegido que cada celda almacene un *byte* que representa la presencia de un pedazo de superficie en un centímetro cúbico, este tipo de arreglos también funciona para almacenar información como color, vectores normales a la superficie medida, o una función implícita que represente la superficie (como la función de distancia con signo de [28]). En estos casos, sin embargo, el uso de la memoria aumenta y su tamaño, respecto del arreglo de ocupación, es el triple en el caso del color, el duodécuplo en el caso del vector normal y el óctuplo en el caso de la función implícita.

5.2. Simulación

En este módulo se busca producir un mapa de profundidad D capturado por un sensor representado por la matriz de parámetros intrínsecos K, en una posición y orientación dadas por una matriz de transformación T, dentro del ambiente virtual almacenado en el arreglo tridimensional V. En el sistema propuesto, el mapa de profundidad se obtiene trazando un rayo por cada pixel, desde la posición del sensor y en dirección de la orientación, a través del volumen hasta encontrar una celda ocupada.

Para cada pixel del mapa de profundidad con coordenadas $\mathbf{u} = (u, v)$, medidas desde la esquina superior izquierda de la imagen y positivas en u a la derecha y en v hacia abajo, podemos obtener el rayo que parte del origen de un sistema de referencia tridimensional, pasa por el plano de proyección, y llega a un punto de profundidad 1 con las ecuaciones 5.2, sustituyendo en esas $D(\mathbf{u}) = 1$. Posteriormente, podemos convertir las componentes de ese rayo al sistema de referencia elegido con las ecuaciones 5.3.

Para cumplir el objetivo de este módulo, necesitamos que la dirección del rayo descrito en el párrafo anterior, que ahora llamamos $\mathbf{r}$, refleje una orientación α, β y γ del sensor y que parta desde una posición arbitraria $(x_s, y_s, z_s)^\intercal$ en el sistema de referencia global. Para la primera parte, la orientación arbitraria, utilizamos las matrices de rotación alrededor de los ejes X, Y y Z de la ecuación 5.1, y con esas obtenemos el rayo en la dirección de la orientación como $\mathbf{r} = \mathtt{R}_z\mathtt{R}_y\mathtt{R}_x\mathbf{r}$. Este proceso es equivalente a tomar los puntos del plano de proyección, rotarlos alrededor del origen y tomar el vector a la nueva posición de cada punto del plano de proyección. Para la segunda parte, la posición arbitraria, definimos el origen de nuestro rayo como la posición $\mathbf{p}_s = (x_s, y_s, z_s)^\intercal$ del sensor. Con estos vectores, podemos determinar la posición, en nuestro sistema de referencia, de cualquier punto en la línea $\mathbf{l}(\lambda) = \mathbf{p}_s + \lambda\mathbf{r}$, que representa los puntos que se proyectan en el pixel (u, v).

Con esta línea definida, modificamos, iterativamente, el parámetro λ para avanzar, en cada paso, en la dirección de $\mathbf{r}$, iniciando con el valor más cercano permitido y terminando al encontrar la primera celda ocupada (*voxel* ocupado) que se cruza con la línea o al llegar a un valor máximo permitido. En cada paso se obtiene una nueva posición en coordenadas globales que, para poder usarse y consultar la ocupación de una celda en el arreglo V, debe ser convertida a la posición en índices del arreglo tridimensional. La ecuación 5.5 realiza esa conversión, y con este procedimiento encontramos un punto de una superficie que se proyecta al pixel (u, v). Si se encontró una celda, el parámetro λ indica la distancia del punto $\mathbf{p}_s$ a la celda.

Si se desea conocer la distancia de la celda al plano de proyección, se puede calcular transformando el vector $(0, 1, 0)$ con la matriz de rotación `R` y calculando la distancia de la celda al plano que contiene el punto $\mathbf{p}_s$ y tiene por normal el vector normal calculado.

Todo el proceso de la simulación se resume en el pseudocódigo 2.

5.3. Implementación

La implementación de los dos módulos descritos utiliza el lenguaje de programación C++. Para las partes que pueden realizarse en paralelo utiliza la plataforma CUDA. A continuación describimos el primer módulo, que para facilitar la comprensión se ha descompuesto en cuatro fases; y el segundo módulo, que con el mismo propósito se ha descompuesto en dos fases.

El primer módulo, la construcción del ambiente, consta de cuatro fases: la adquisición de la posición y la orientación del sensor, así como la adquisición del mapa de profundidad en ese momento; el procesamiento del mapa de profundidad para obtener una lista de puntos en el sistema de referencia global; la actualización del modelo de *voxels*; y la escritura a disco duro de la representación del ambiente generada. Las tres primeras fases se realizan para cada cuadro del mapa de profundidad disponible, y la última cuando no hay más que procesar.

Para la primera fase, la adquisición del mapa de profundidad se realiza en el CPU con la ayuda de la bilbioteca *OpenNI*, y la posición y orientación se lee de un archivo de texto. Con el propósito de abstraer estas dos operaciones y de cambiar, en el futuro, la forma de adquisición y posición, se definen las clases `FrameGrabber` y `PositionGrabber`. Cada una de estas clases, y a través de un objeto de cada una de ellas, permite obtener un cuadro del mapa de profundidad y una posición fácilmente con funciones `getFrame()` y `getPosition()`.

Pseudocódigo 2: Obtención de un mapa de profundidad simulado, mediante trazado de rayos, a partir de la información almacenada en un arreglo tridimensional de ocupación.

input : La representación del ambiente como un arreglo tridimensional $V(i_x, i_y, i_z)$. La matriz de parámetros intrínsecos K del sensor, la matriz de rotación R (al rededor de los ejes X,Y y Z) que representa la orientación del sensor con los ángulos de rotación γ,β y α. La posición $\mathbf{p}_s$ del sensor en el sistema de referencia global. La escala s necesaria para que la parte entera de las coordenadas corresponda a la unidad de longitud elegida.

output: El mapa de profundidad simulado D

```
1  foreach pixel u = (u, v) del mapa de profundidad simulado D in parallel do
2      r    sK^{-1}(u, 1)^T
3      r      Rr
4      r      r      r
5      λ    m´nima distancia a veri car.
6      inc    incremento, en unidades de longitud, para cada paso.
7      pos    p_s + λr
8      (i_x, i_y, i_z)    ndicesDelArreglo(pos)
9      impacto    falso
10     while la celda (i_x, i_y, i_z) esté dentro de los l´mites del arreglo and λ <máxima
       distancia a veri car and impacto = verdadero do
11         if V(i_x, i_y, i_z) está ocupada then
12             impacto    verdadero
13         else
14             λ    λ + inc
15             pos    p_s + λr
16             (i_x, i_y, i_z)    ndicesDelArreglo(pos)
17     if impacto = verdadero then
18         D(u, v)    λ
19     else
20         D(u, v)    0
```

Para la segunda fase, la implementación en CUDA utiliza una configuración de lanzamiento bidimensional de manera que cada hilo trabaja con la información de un pixel del mapa de profundidad. El mapa de profundidad es un arreglo de enteros de 16 *bits* (`unsigned short`) que se copia de memoria del anfitrión a memoria global del dispositivo. La matriz de transformación $\mathtt{T}_g$ es un arreglo en memoria constante, igual que los parámetros c_x, c_y, f_x y f_y de la matriz de parámetros intrínsecos K. El resultado final, o sea, las coordenadas de los puntos en el sistema global, se almacena en tres arreglos `X`, `Y`, y `Z` de números de precisión simple de 32 *bits*, y cada elemento corresponde al pixel del mapa de profundidad

que produjo ese punto. En el apéndice A.2 se muestra el código del *kernel* que realiza el procedimiento descrito.

La tercera fase procesa la lista de puntos para obtener los índices del arreglo tridimensional, verifica que estos se encuentren dentro de los límites, y activa la celda correspondiente.

La cuarta fase escribe el arreglo tridimensional como un archivo binario lineal. Recordemos que V_x, V_y y V_z son las las dimensiones del arreglo tridimensional, y sea $\mathbf{v}$ el *voxel* con índices que inician en cero en el arreglo tridimensional (i_x, i_y, i_z), la posición de $\mathbf{v}$ en el archivo binario está dada por el polinomio de direccionamiento $V_y V_x i_z + V_x i_y + i_x$.

El segundo módulo, la simulación, consta de dos fases: cargar la representación del ambiente desde un archivo binario en disco duro; y simular para una posición y orientación un cuadro del mapa de profundidad.

La primera fase carga el archivo que contiene el arreglo tridimensional y lo pone a disposición de la clase `FrameGrabber`.

Para la segunda fase, la clase `FrameGrabber` contiene una función `getFrame` que recibe una posición y devuelve un cuadro de un mapa de profundidad simulado. Para obtenerlo, esta función ejecuta la implementación en CUDA del trazado de rayos descrito en la sección anterior. Utiliza una configuración de lanzamiento bidimensional de manera que cada hilo generará la información de un pixel del mapa de profundidad. El arreglo tridimensional es un arreglo de enteros de 8 *bits* que se copia de memoria del anfitrión a memoria global del dispositivo. La matriz de transformación $\mathtt{T}_g$, que representa la posición y orientación del sensor a simular, es un arreglo en memoria constante, igual que los parámetros c_x, c_y, f_x y f_y de la matriz de parámetros intrínsecos K. El resultado final, es decir, el mapa de profundidad simulado, es un arreglo de enteros de 16 *bits* (`unsigned short`) que almacena la imagen por filas empezando por la esquina superior izquierda. En el apéndice A.3 se muestra el código del *kernel* que realiza el procedimiento descrito.

Cap´tulo 6

Experimento y resultados

Para probar el sistema descrito en el capítulo anterior, realizamos el experimento que se describe a continuación y cuyos resultados también se incluyen en este capítulo.

Con el robot humanoide del Laboratorio de Biorobótica, Justina, y con el sensor Kinect que está montado en su cabeza, capaz de girar en los ejes Z y X descritos anteriormente y cuya matriz de parámetros intrínsecos K ya era conocida, capturamos doce videos de la siguiente manera: capturamos la orientación inicial; luego hicimos que girara un radián, en sentido antihorario, alrededor del eje Z (que volteara a la izquierda); medio radián, en sentido horario, al rededor del eje X (que viera hacia abajo); con esta última rotación, giró un radián, en sentido horario, alrededor del eje Z (que regresara a la orientación inicial pero viendo hacia abajo); de ahí, un radián, en sentido horario, alrededor del eje Z (que viera a la derecha y abajo); medio radián, en sentido antihorario, al rededor del eje X (que viera a la derecha a la altura de las dos primeras mediciones); y repetimos estas seis mediciones después de indicarle al robot que se moviera un metro hacia adelante. Como un vector de posición (x, y, z) y uno de orientación (α, β, γ) lo anterior se muestra en la tabla 6.1.

A partir de estas doce mediciones generamos, utilizando el módulo de construcción, una representación del ambiente con $1024 \times 1024 \times 480$ celdas tridimensionales en las direcciones V_x, V_y y V_z.

Con esa representación generamos, utilizando el módulo de simulación, el mapa de profundidad en diferentes posiciones y orientaciones. Incluyendo aquellas en las que se capturó originalmente la información, para poder realizar una comparación entre ambas mediciones.

En este capítulo presentamos, primeramente, algunas observaciones de la optimización los dos módulos; después, comparamos el desempeño del sistema al generar la representación

del ambiente y el mapa de profundidad contra una versión en CPU optimizada mediante la bandera `-O3` del compilador; finalmente, presentamos información del error relativo de las mediciones simuladas respecto a las mediciones originales del sensor.

Todos los cálculos se realizaron en una computadora de escritorio con un procesador Intel Core i7-2600K CPU @ 3.40GHz × 8, una GPU Nvidia GeForce GTX 680, y 16 GB de memoria RAM.

Medición	**Posición [cm]** (x, y, z)	**Orientación [rad]** (α, β, γ)
1	$(0, 0, 0)$	$(0, 0, 0)$
2	$(0, 0, 0)$	$(1, 0, 0)$
3	$(0, 0, 0)$	$(1, 0, -0.5)$
4	$(0, 0, 0)$	$(0, 0, -0.5)$
5	$(0, 0, 0)$	$(-1, 0, -0.5)$
6	$(0, 0, 0)$	$(-1, 0, 0)$
7	$(0, 100, 0)$	$(0, 0, 0)$
8	$(0, 100, 0)$	$(1, 0, 0)$
9	$(0, 100, 0)$	$(1, 0, -0.5)$
10	$(0, 100, 0)$	$(0, 0, -0.5)$
11	$(0, 100, 0)$	$(-1, 0, -0.5)$
12	$(0, 100, 0)$	$(-1, 0, 0)$

Tabla 6.1: Posiciones y orientaciones del sensor para el experimento.

6.1. Consideraciones de optimización

La herramienta presentada en la sección 2.5.5, el perfilador visual de NVIDIA, resultó de gran utilidad para optimizar los *kernels* de cada módulo.

Para el primero sugirió, después de ejecutarlo varias veces, que se redujera el número de registros utilizados (utilizando la bandera `-maxrregcount` del compilador) por hilo para aumentar la ocupación de los *SM* y con ello, el desempeño del *kernel*. Siguiendo el consejo de la herramienta, la ocupación pasó del 70 % al 91 %, y el tiempo promedio al procesar un cuadro del mapa de profundidad y convertirlo al sistema de referencia global mejoró de 300 s a 200 s. La siguiente optimización, que no fue sugerida por el perfilador porque tenía que ver con la lógica del programa, consistió en escribir, directamente, los parámetros de la matriz intrínseca K en el código, igual que la escala utilizada; y en eliminar el salto

(*stride*) que un hilo tendría que realizar si se lanzan menos hilos que pixeles del mapa de profundidad. Con este último cambio, el tiempo promedio de procesamiento por cuadro fue de 170 s.

El segundo *kernel*, después de analizarlo con el perfilador, fue optimizado de manera similar. Con la bandera `-maxrregcount` establecimos el máximo de registros, eliminamos las lecturas a memoria constante de la matriz de parámetros intrínsecos, y quitamos la lógica del salto (*stride*). Con lo anterior, logramos pasar de una ocupación del 68 % a una del 93 %; y de generar un cuadro en 20 ms a generarlo en 16 ms. En apariencia no es una gran mejora, pero permite generar diez cuadros más por segundo, es decir, pasar de 50 a 60 fps (*frames per second*).

6.2. Comparación contra CPU

En la tabla 6.2 se muestra, para cada una de las mediciones de la tabla 6.1 y para el total de ellas, el desempeño de la conversión, en CPU y GPU, del mapa de profundidad a puntos del sistema de referencia global elegido. La segunda columna muestra el tiempo de procesamiento en CPU, la tercera el tiempo de procesamiento en GPU, y la cuarta el tiempo de procesamiento en GPU más el tiempo que toma transferir el mapa de profundidad al GPU y los resultados de vuelta al sistema anfitrión. Cada medición se procesó cien veces, por lo que se muestra el promedio y, entre paréntesis, la varianza.

Realizando solamente la conversión del mapa de profundidad a puntos en el sistema de referencia, la aceleración ($T_{secuencial}$ $T_{paralelo}$) es de 72; y añadiendo las transferencias de memoria, de 3.8.

La tabla 6.3 muestra el desempeño de la simulación de un mapa de profundidad en CPU y en GPU. La implementación en CPU es una adaptación directa del código mostrado en el apéndice A.3. Para este reporte, generamos 12 000 cuadros simulados y para cada uno medimos el tiempo requerido para su procesamiento en CPU y en GPU. Se muestra, en la primera fila, el tiempo que tomó todo el procesamiento; en la segunda, el tiempo promedio de procesamiento para generar un cuadro y, entre paréntesis, la varianza. La figura 6.2 muestra los histogramas del tiempo de ejecución. En la simulación de un mapa de profundidad la aceleración resultó de 25.6.

A manera de comparación gráfica, en la figura 6.1 se muestra el tiempo de procesamiento, en minutos, en función del número de cuadros a procesar, para el CPU (en azul) y para el GPU (en verde).

Medición	**CPU** $[ms]$	**GPU** $[ms]$	**GPU+TDM** $[ms]$
1	7.8 (2.06)	0.1512 (1.374×10^{-8})	3.54 (0.1024)
2	11.0 (9.17)	0.1446 (5.1×10^{-4})	3.48 (0.0285)
3	13.16 (6.88)	0.1478 (1.737×10^{-7})	3.51 (0.1164)
4	8.12 (3.06)	0.1541 (2.77×10^{-7})	3.51 (0.00859)
5	11.2 (3.56)	0.1366 (7.67×10^{-8})	3.52 (0.0375)
6	6.52 (3.77)	0.1325 (2.5×10^{-6})	3.49 (0.00941)
7	8.2 (1.738)	0.1538 (7.1×10^{-8})	3.51 (0.01271)
8	11.2 (10.34)	0.1422 (2.35×10^{-7})	3.49 (0.0313)
9	13.56 (7.08)	0.1499 (5.27×10^{-4})	3.46 (0.0390)
10	13.64 (5.85)	0.1546 (7.36×10^{-5})	3.54 (0.01174)
11	11.92 (3.87)	0.1435 (5.31×10^{-4})	3.51 (0.01358)
12	10.64 (5.57)	0.1322 (6.91×10^{-8})	3.47 (0.01025)
Tiempo total	127.0	1.743	42.0
Tiempo promedio por cuadro	10.58 (5.78)	0.1453 (6.59×10^{-5})	3.58 (7.4266×10^{-4})

Tabla 6.2: Resultados de las comparaciones.

Medición	**CPU**	**GPU**
Tiempo para generar 12 000 cuadros	82.45 min	3.22 min
Tiempo promedio para generar un cuadro (varianza)	412 ms (6.74)	16.10 ms (0.0069)
Cuadros por segundo	2.42	62.1

Tabla 6.3: Comparación del rendimiento entre CPU y GPU.

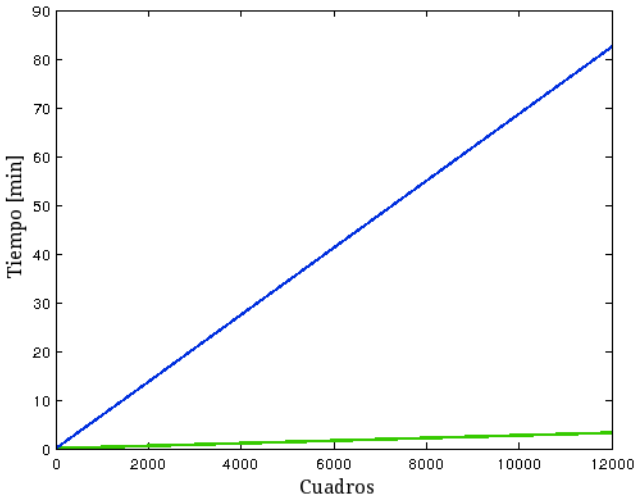
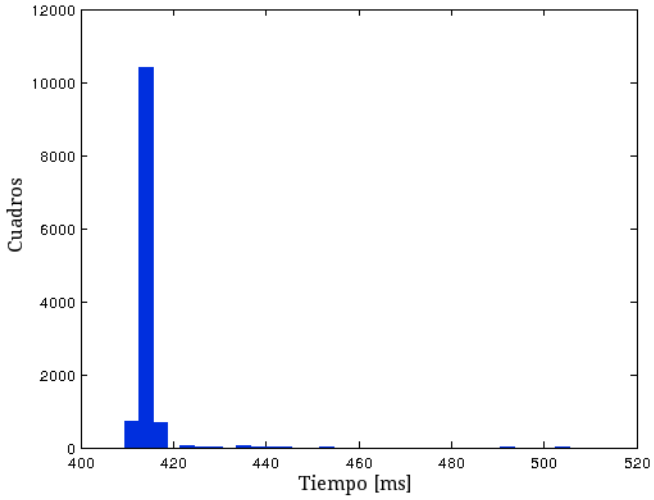
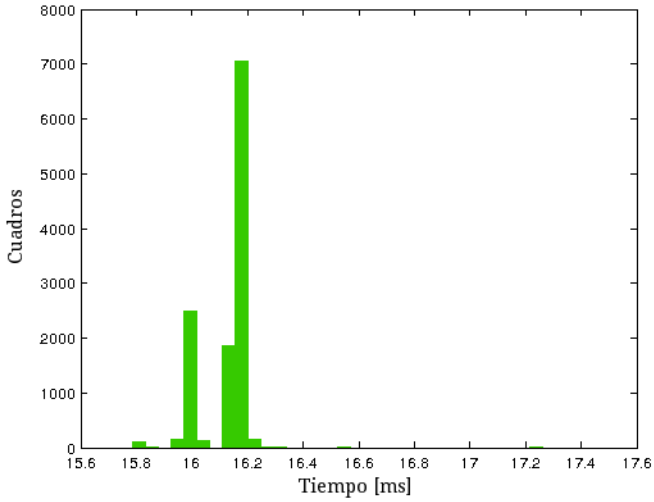


Figura 6.1: Tiempo de ejecución CPU y GPU



(A) Histograma del tiempo de ejecución en CPU de 12 000 muestras.



(B) Histograma del tiempo de ejecución en GPU de 12 000 muestras.

FIGURA 6.2: Histogramas del tiempo de ejecución por cuadro.

6.3. Comparación de medidas

Comparamos las medidas de un mapa de profundidad A, simulado con el sistema (fig. 6.3 A), con las medidas del mapa de profundidad B, obtenidas con el sensor Kinect (fig. 6.3 B). La comparación, que se muestra en la tabla 6.4, se realizó para cada una de las posiciones y orientaciones de la tabla 6.1.

En la segunda columna mostramos el porcentaje de pixeles del mapa A cuya medida tiene, respecto a la medida correspondiente en B, un error relativo mayor del 10 %. Es decir, el porcentaje de pixeles $\mathbf{u}$ para los que $\frac{A(\mathbf{u})-B(\mathbf{u})}{B(\mathbf{u})} > 0.1$.

En la tercera columna mostramos, en porcentaje, la razón del número de pixeles válidos nuevos (que tienen medida en A, pero no en B) al número de pixeles válidos en B. Esto es, si A_v es el número de pixeles válidos de A y B_v el número de pixeles válidos de B, mostramos $100 \times \frac{A_v - B_v}{B_v}$. Con esto podemos decir, por ejemplo, que el mapa A tiene 22 % más pixeles válidos que el mapa B.

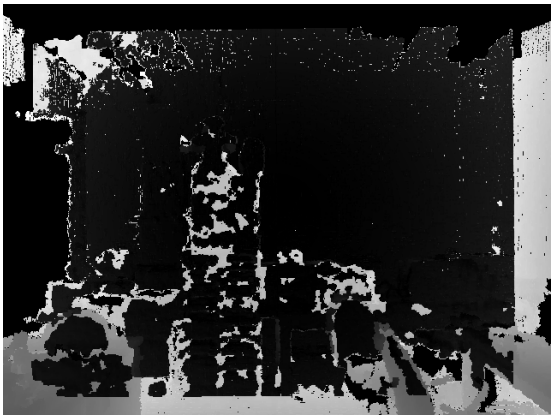
Medición	**Porcentaje de pixeles con error relativo mayor al 10 %**	**Porcentaje de pixeles nuevos**
1	2.53	26.4
2	3.34	22.0
3	10.26	31.3
4	5.08	32.0
5	7.5	37.1
6	4.3	21.7
7	6.72	33.7
8	7.67	46.0
9	31.2	65.7
10	10.23	37.9
11	16.93	59.5
12	7.92	32.0
Promedio	9.47	37.1

Tabla 6.4: Resultados de las comparaciones.

Al analizar la tabla, se observa un incremento en el error para las medidas realizadas un metro más adelante. Esto llevó a suponer que el robot no se movió 100 *cm* como le indicamos. La suposición se verificó al analizar los mapas de profundidad obtenidos con Kinect en las posiciones 1 y 7 de la tabla 6.1, y se observó que el robot se movió, aproximada y solamente, 84 *cm*. Con esta información generamos, nuevamente, el modelo del ambiente, pero sin tomar en cuenta la información capturada por el Kinect en las posiciones y orientaciones 7 a 12 de la tabla 6.1. Utilizando ese nuevo modelo y sabiendo que se movió 84 *cm*, simulamos mapas

(A) Medición simulada con el sistema ejemplo. A.

(B) Medición realizada con Kinect. B.



(C) Diferencia entre las medidas. $|A - B|$.

FIGURA 6.3: Mediciones comparadas.

de profundidad en posiciones y orientaciones similares a las de la tabla 6.1, pero variando la posición a la aproximación encontrada. Realizamos la misma comparación que antes y los resultados se muestran en la tabla 6.5.

Medición	**Porcentaje de pixeles con error relativo mayor al 10 %**	**Porcentaje de pixeles nuevos**
1	1.699	16.20
2	1.641	9.27
3	8.67	26.4
4	2.55	24.3
5	3.69	29.9
6	2.73	13.86
7	6.08	21.6
8	10.51	6.12
9	32.4	37.2
10	4.98	18.79
11	11.23	33.4
12	2.82	14.92
Promedio	7.42	21.0

Tabla 6.5: Resultados de las comparaciones con las modificaciones de las posiciones de simulación.

Como se esperaba, eliminar la mitad de las mediciones hechas con Kinect tuvo como consecuencia indeseada la reducción del porcentaje de pixeles nuevos. También se redujo, en promedio, el porcentaje de pixeles con error relativo mayor al 10 %. A pesar de esta última y deseada reducción, todavía quedan números alarmantes como el de la novena medición. En esta, se le pidió al robot que girara la cabeza a la izquierda $1\,rad$ y luego hacia abajo $0.5\,rad$, y el alto error relativo sugiere que no giró lo indicado, o que la distancia entre el eje de giro del motor sobre el que está montado el sensor del robot y el eje X alrededor del cual aplicamos la rotación no es despreciable y, en ese caso, las rotaciones al rededor del eje X debieron ocurrir después de traslación de esta distancia. Cualquiera de los dos casos contribuye a las conclusiones del capítulo siguiente, en donde discutimos los inconvenientes y dificultades de depender de las mediciones de ángulos y desplazamientos del robot.

Cap´tulo 7

Conclusiones

Este trabajo tuvo por objetivo principal desarrollar un sistema heterogéneo (CPU-GPU) que permitiera, en primer lugar, construir una representación del ambiente de un robot con un sensor RGB-D montado, utilizando la posición, orientación y mapas de profundidad provistos por el robot; y, después, simular lo que el sensor de profundidad capturaría en posiciones y orientaciones arbitrarias.

La decisión de hacer un sistema heterogéneo, y no solamente con procesamiento en CPU, radicó en que, de unos años a la fecha, los procesadores han disminuido su crecimiento en velocidad, para crecer ahora, más bien, en número de núcleos, por lo que procesar en paralelo ya es cada día menos una opción y más una necesidad. Como herramienta para el procesamiento en paralelo escogimos las GPU y, particularmente, las tarjetas de *NVIDIA* por su plataforma CUDA. Aunque existen otras tecnologías, como OpenCL, CUDA ofrece una manera accesible de aprovechar la gran cantidad de procesadores de las GPU. En este trabajo presentamos un resumen de las ideas más importantes para utilizar la plataforma, esperando que sirva como una referencia rápida o una introducción al tema. El vistazo rápido a las arquitecturas permite observar y concluir que las mejoras en *hardware* son continuas, y que podemos esperar mayor capacidad de procesamiento y memoria disponible.

La información de profundidad puede provenir, como analizamos en este trabajo, de dispositivos con diferentes métodos de adquisición, pero al final proporcionan un arreglo en el que cada elemento contiene la distancia del sensor al área representada por ese elemento. No es novedad que el sensor con que trabajamos, Kinect, presenta pérdida de información en varias zonas y ruido en las mediciones. En un futuro no tan lejano, no obstante, la información será más precisa y refinada, pues los sensores RGB-D o de profundidad también evolucionan. En los sistemas de visión estereoscópica, por ejemplo, las cámaras tienen mejor resolución y, por tanto, los mapas de profundidad generados también. Otro caso, la

tecnología *ToF* se ha hecho más accesible, y la segunda versión del Kinect ya cuenta con una cámara de ese tipo, capaz de proporcionar mapas de casi el doble de resolución y con una calidad que, a la vista, los hace parecer más una reconstrucción, como la hecha por los sistemas presentados en el cuarto capítulo, que un mapa de profundidad. Así que, como con las GPU, podemos contar con mejoras en el *hardware* y en la calidad de la información de entrada.

Las mejoras de las GPU y de los sensores RGB-D facilitarán el trabajo y algunos de los problemas a atacar al construir un ambiente a partir de mapas de profundidad y al simular un sensor en ese ambiente. Por ejemplo, si la memoria de las primeras crece, el aumento de tamaño de las escenas representadas no será tan preocupante, porque habrá suficiente espacio para guardar la información. También, si la precisión de las medidas de los sensores aumenta, el procesamiento sobre los mapas de profundidad para mejorarlos no será tan intensivo. Existen, sin embargo, otras áreas que seguirán requiriendo atención y que no se solucionan simplemente por las mejoras en la tecnología. El sistema desarrollado en este trabajo, presentado en el capítulo cinco, nos permitió ganar entendimiento de algunas de esas áreas.

Desde el comienzo del diseño nos propusimos llevar el problema a su forma más simple, entenderlo bien, dar una solución, y descubrir las dificultades prácticas que se presentan.

Con esa simplificación en mente, el problema de construir una representación del ambiente se resume en dos pasos: convertir los elementos del mapa de profundidad a puntos tridimensionales; y en convertir esos puntos —con una matriz de transformación que incluye una rotación (orientación del sensor) y una traslación (posición)— a un mismo sistema de referencia y almacenarlos de alguna manera. En el sistema desarrollado, el primer paso se realizó con la matriz de parámetros intrínsecos K del sensor y la ecuación 3.7. El segundo, se llevó a cabo tomando por buenas las mediciones, proporcionadas por el robot, del giro y desplazamiento del sensor, y utilizando esos ángulos y posiciones generamos la matriz de la ecuación 3.8 para referir todos los puntos en un mismo sistema y los almacenamos en un arreglo tridimensional en el que cada celda representa un centímetro cúbico.

Siguiendo con la simplificación, el problema de simular un sensor dentro de un ambiente virtual se resume en visualizar la información almacenada, transformando una cámara con los mismos parámetros intrínsecos que el sensor original y que observa desde la posición y en la orientación simulada. En el sistema desarrollado, trazamos un rayo, por pixel, desde la posición del sensor y en dirección a la posición del pixel en el plano de proyección, y avanzamos este rayo hasta encontrar una celda ocupada.

Con el sistema implementado, el capítulo anterior presentó, resumidamente, dos resultados. Primero, que el procesamiento en GPU es mejor opción que el procesamiento en

CPU en nuestro sistema, puesto que el sistema heterogéneo pudo procesar, en el módulo de la construcción, 3.8 veces más cuadros por segundo que el sistema en CPU, y 25 veces más cuadros por segundo en el módulo de la construcción. Y, segundo, que la construcción del mundo y, por tanto, de los mapas de profundidad simulados están lejos de ser óptimos, como lo prueba la gran cantidad de pixeles simulados cuya medida tiene más del 10 % de error relativo a la medida del sensor. Del primer resultado entendemos que los esfuerzos futuros en la construcción y simulación de ambientes deberán emplear la GPU para tener buen rendimiento. Del segundo, que depender de las mediciones del robot para obtener la matriz de transformación que convierte los puntos a un mismo sistema de referencia sumado a la mala calidad de los mapas de profundidad, puede provocar la construcción de representaciones paupérrimas del entorno del robot. La conclusión anterior, aunque pesimista en apariencia, produjo preguntas como qué se puede hacer para mejorar los mapas de profundidad de entrada? O, si no se tiene la información de las rotaciones y la posición del sensor, cómo se obtiene la transformación que convierte los puntos a un mismo sistema de referencia? También, será un arreglo tridimensional de ocupación una buena manera de guardar la información, o habrá otras?

Una solución a todas esas interrogantes fue encontrada en el sistema analizado en el capítulo cuarto. Ese análisis pretende ser esclarecedor para quienes quieran incursionar en este problema en un futuro cercano. Además, intenta sugerir las áreas del problema en que se puede trabajar. Brevemente, el procesamiento de los mapas de profundidad para reducir la falta de información y el ruido de los sistemas actuales, el diseño de estructuras de datos que permitan almacenar representaciones de mayor tamaño y acceder a la información eficientemente en la GPU, los algoritmos para obtener la transformación que refiere todos los puntos a un mismo sistema que serán siempre necesarios y desafiantes, y las técnicas para trabajar con escenas dinámicas.

La conclusión, final y personal, es que este trabajo aporta al entendimiento del problema, y que funcionará como un compendio de conocimientos y referencias útiles para los nuevos estudiantes y sus tutores, particularmente los hispanohablantes y del futuro cercano, que deseen trabajar en la construcción y simulación de ambientes virtuales con GPU y sensores RGB-D.

7.1. Trabajo a futuro

Para mejorar la construcción del ambiente virtual hay que eliminar la dependencia de las mediciones de posición y orientación que da el robot. Convendría tener, como entrada única del sistema, y aunque se decida hacerlo en tiempo real o no, la secuencia de los mapas de

profundidad, y tener un flujo de trabajo similar al del capítulo cuarto, al de alguno de los sistemas más recientes, o un sistema que supere al del estado actual diseñado, más bien, por un aspirante a doctor y sus asesores.

En cualquiera de los tres casos, hará falta dividir el trabajo para tener una implementación. Si, por ejemplo, se tomara el sistema del capítulo cuatro, alguien podría encargarse del preprocesamiento de los mapas de profundidad y conversión del mapa a puntos y vectores normales; alguien más, de la estimación de la transformación que refiere todos los puntos a un mismo sistema de referencia; otro más, de actualizar la TSDF; y uno último, de generar los puntos y normales que se utilizan para estimar la transformación, que es, además, la parte de simular el sensor dentro del ambiente.

Existen opciones para no empezar de cero. La biblioteca PCL [43] ofrece todo un marco de trabajo que permite obtener mapas de profundidad de varios sensores, convertirlos a conjuntos de puntos tridimensionales, manipularlos, segmentarlos, almacenarlos en estructuras jerárquicas, e inclusive tiene un módulo para generar una TSDF como la utilizada por KinectFusion. Algunos grupos de trabajo [44, 45] tienen sus códigos disponibles en línea y se pueden utilizar si se desean resultados inmediatos o para comparar los resultados.

Otros temas interesantes con GPU y sensores RGB-D pueden consultarse en [46], donde se presentan algunos trabajos recientes en áreas como reconocimiento de gestos, personas, objetos, etc.

Apéndice A

Código fuente

A.1. Código fuente del ejemplo del cap´tulo segundo

Ejemplo del capítulo dos. Consiste en obtener el mínimo de una función evaluada en un conjunto de valores de entrada y también el valor que lo produce. El arreglo de entrada se procesa primero en la GPU y la reducción final se lleva a cabo en la CPU. El *kernel* en GPU se encarga, primero, de evaluar para cada elemento del arreglo de entrada una función $f(x)$; después, cada bloque de hilos CUDA encuentra el mínimo dentro de este bloque; y, finalmente, el hilo maestro de cada bloque escribe el resultado a memoria global. La función en CPU se encarga de reducir los resultados producidos por cada bloque.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #define TPB 1024
5  #define NB 1

7  __device__ float f(float x){
8    return (x-1)*(x-1);
9  }

11 __global__ void functionMin(float* array,
12                             float* minPerBlock, int* minPerBlockIdx, int N){

14   __shared__ float minSM[TPB];
15   __shared__ int minIdxSM[TPB];

17   for(int globalIdx=threadIdx.x+blockDim.x*blockIdx.x;
18           globalIdx<N; globalIdx+=gridDim.x*blockDim.x){
19     minSM[threadIdx.x]=f(array[globalIdx]);
20     minIdxSM[threadIdx.x]=globalIdx;
21     __syncthreads();
```

```
23      for(int activeThreads=blockDim.x>>1;
24              activeThreads>0; activeThreads>>=1){
25        if(threadIdx.x < activeThreads){
26          minSM[threadIdx.x]=fmin(minSM[threadIdx.x],
27                                  minSM[threadIdx.x+activeThreads]);
28          if(minSM[threadIdx.x]==minSM[threadIdx.x+activeThreads]){
29            minIdxSM[threadIdx.x]=minIdxSM[threadIdx.x+activeThreads];
30          }
31        }
32        __syncthreads();
33      }

35      if(threadIdx.x==0){
36        minPerBlock[blockIdx.x +
37               (globalIdx/(gridDim.x*blockDim.x))*gridDim.x]=minSM[threadIdx.x];
38        minPerBlockIdx[blockIdx.x +
39                (globalIdx/(gridDim.x*blockDim.x))*gridDim.x]=minIdxSM[threadIdx.x];
40      }
41    }
42  }

44  int cudaCheckError(cudaError_t err){
45    if (err != cudaSuccess) {
46        fprintf(stderr, "Error %s\n", cudaGetErrorString(err));
47    }
48    return err;
49  }

51  float* array(float start, float end, int N, int extra){
52    float *buffer;
53    buffer=(float *)malloc(sizeof(float)*(N+extra));
54    for(int i=0;i<N;i++){
55      buffer[i]=start+i*(end-start)/N;
56    }
57    for(int i=N;i<N+extra;i++){
58      buffer[i]=NAN;
59    }
60    return buffer;
61  }

63  void min(float *array, int *idx, float *minValue, int *minIdx, int N){

65    (*minValue)=array[0];
66    (*minIdx)=idx[0];

68    for (int i=1;i<N;i++){
69      (*minValue)=fminf((*minValue),array[i]);
70      if((*minValue)==array[i]) (*minIdx)=idx[i];
71    }
72  }
```

```
74 int main(void) {

76   float *dataHost = NULL;
77   const int N=1024*90+6;
78   const int extra=(N%TPB!=0)?TPB-N%TPB:0;
79   const int numberOfBlocks=N/TPB+(N%TPB != 0);
80   dataHost=array(-2.0f, 2.0f, N, extra);

82   float *dataDevice;
83   float *minPerBlockDevice;
84   int *minPerBlockIdxDevice;
85   cudaError_t e=cudaSuccess;
86   e=cudaMalloc(&dataDevice,sizeof(float)*(N+extra));
87     if (cudaCheckError(e)) exit(EXIT_FAILURE);
88   e=cudaMalloc(&minPerBlockDevice,sizeof(float)*(numberOfBlocks));
89     if (cudaCheckError(e)) exit(EXIT_FAILURE);
90   e=cudaMalloc(&minPerBlockIdxDevice,sizeof(int)*(numberOfBlocks));
91     if (cudaCheckError(e)) exit(EXIT_FAILURE);

93   e=cudaMemcpy(dataDevice, dataHost, sizeof(float) * (N+extra),
       cudaMemcpyHostToDevice);
94     if (cudaCheckError(e)) exit(EXIT_FAILURE);

96   dim3 grid(NB,1,1);
97   dim3 block(TPB,1,1);
98   functionMin<<<grid, block>>>(dataDevice, minPerBlockDevice, minPerBlockIdxDevice
       , N+extra);
99   e=cudaDeviceSynchronize();
100    if (cudaCheckError(e)) exit(EXIT_FAILURE);
101  e=cudaGetLastError();
102    if (cudaCheckError(e)) exit(EXIT_FAILURE);

104  float *minPerBlockHost;
105  int *minPerBlockIdxHost;
106  minPerBlockHost=(float *)malloc(sizeof(float)*(numberOfBlocks));
107  minPerBlockIdxHost=(int *)malloc(sizeof(int)*(numberOfBlocks));
108  e=cudaMemcpy(minPerBlockHost, minPerBlockDevice, sizeof(float) * numberOfBlocks,
        cudaMemcpyDeviceToHost);
109    if (cudaCheckError(e)) exit(EXIT_FAILURE);
110  e=cudaMemcpy(minPerBlockIdxHost, minPerBlockIdxDevice, sizeof(int) *
       numberOfBlocks, cudaMemcpyDeviceToHost);
111    if (cudaCheckError(e)) exit(EXIT_FAILURE);

113  e=cudaFree(minPerBlockDevice); if (cudaCheckError(e)) exit(EXIT_FAILURE);
114  e=cudaFree(minPerBlockIdxDevice); if (cudaCheckError(e)) exit(EXIT_FAILURE);
115  e=cudaFree(dataDevice); if (cudaCheckError(e)) exit(EXIT_FAILURE);

117  float minX;
118  float minFofX;
119  int minIdx;
120  min(minPerBlockHost, minPerBlockIdxHost, &minFofX, &minIdx, numberOfBlocks);
```

```
121   minX=dataHost[minIdx];
122   printf("Minimum value is at idx=%u, x=%f and f(x)=%f\n", minIdx, minX, minFofX);

124   return 0;
125 }
```

Listado A.1: Ejemplo de un programa en CUDA C.

A.2. Conversión de un mapa de profundidad a puntos en un sistema cartesiano

```
__constant__ float parameters[4]={3.1859233711350760e+002,2.4926027230995192e
    +002,5.2338812559311623e+002,5.2332543643433257e+002};  //cX cY fX fY
__constant__ float cameraToWorldMatrix[16];
__constant__ float scale=10.0f;

__global__ void convert2World(unsigned short* depthMap,float *x,float *y, float *z
    , int w, int h){
    float xW=NAN,yW=NAN,zW=NAN;
    unsigned short depthValue;
    for(int globalTidY=threadIdx.y+blockIdx.y*blockDim.y;globalTidY<h;globalTidY+=
    gridDim.y*blockDim.y){
      for(int globalTidX=threadIdx.x+blockIdx.x*blockDim.x;globalTidX<w;globalTidX
    +=gridDim.x*blockDim.x){
        depthValue=depthMap[globalTidY*w+globalTidX];
        if(depthValue>0){
          float xWTmp=(globalTidX-parameters[0])* depthValue / (parameters[2]*
    scale);
          float yWTmp=depthValue/scale;
          float zWTmp=-1.0f*(globalTidY-parameters[1])* depthValue / (parameters
    [3]*scale);

          xW= cameraToWorldMatrix[0]*xWTmp + cameraToWorldMatrix[1]*yWTmp
              + cameraToWorldMatrix[2]*zWTmp + cameraToWorldMatrix[3];
          yW=cameraToWorldMatrix[4]*xWTmp + cameraToWorldMatrix[5]*yWTmp
              + cameraToWorldMatrix[6]*zWTmp + cameraToWorldMatrix[7];
          zW=cameraToWorldMatrix[8]*xWTmp + cameraToWorldMatrix[9]*yWTmp
              + cameraToWorldMatrix[10]*zWTmp + cameraToWorldMatrix[11];
        }

        x[globalTidY*w+globalTidX]=xW;
        y[globalTidY*w+globalTidX]=yW;
        z[globalTidY*w+globalTidX]=zW;
      }
    }
}
```

LISTADO A.2: *Kernel* que convierte las profunidades de un mapa a puntos en un sistema cartesiano tridmensional.

A.3. Obtención de un mapa de profundidad a partir de una representación del ambiente

```
__constant__ float parameters[4]={3.1859233711350760e+002,2.4926027230995192e
    +002,5.2338812559311623e+002,5.2332543643433257e+002};  //cX cY fX fY
__constant__ float scale=1.0f;
__constant__ float cameraPosition[16];//Matrix used to transform points on
    projection plane to camera position

__global__ void renderDepthMap( volumeType *volume,
                  int width, int height, //Separation between pixels in length
    units, the width and size of the image.
                   depthMapType *depthMap){ //depthMap
  float eyeX,eyeY,eyeZ; //vector to the camera position
  float vX,vY,vZ; //vector from  eye  to the point of the projection plane that
    each thread will process.
  float normV;
  float nX,nY,nZ; //normal vector to projection plane
  float stepX,stepY,stepZ;
  float posX,posY,posZ;
  float aux;
  const int maxSteps = 500; //maximum number of steps to march ray
  bool hit;

  for (int globalTidY=threadIdx.y+blockIdx.y*blockDim.y;globalTidY<height;
    globalTidY+=gridDim.y*blockDim.y){
    for(int globalTidX=threadIdx.x+blockIdx.x*blockDim.x;globalTidX<width;
    globalTidX+=gridDim.x*blockDim.x){

      //initialize pos to the point of the projection plane that each thread will
    process
      posX=(globalTidX-parameters[0]) / (parameters[2]*scale);
      posY=1;
      posZ=-1.0f*(globalTidY-parameters[1]) / (parameters[3]*scale);

      //Transform points of virtual projection plane to match the position and
    orientation of the camera.
      vX= cameraPosition[0]*posX + cameraPosition[1]*posY
        + cameraPosition[2]*posZ + cameraPosition[3];
      vY= cameraPosition[4]*posX + cameraPosition[5]*posY
        + cameraPosition[6]*posZ + cameraPosition[7];
      vZ=cameraPosition[8]*posX + cameraPosition[9]*posY
         + cameraPosition[10]*posZ + cameraPosition[11];

      eyeX=cameraPosition[3];
      eyeY=cameraPosition[7];
      eyeZ=cameraPosition[11];

      vX-=eyeX;
      vY-=eyeY;
```

```
40        vZ-=eyeZ;
41        normV=sqrtf(vX*vX+vY*vY+vZ*vZ);
42        vX/=normV; vY/=normV; vZ/=normV;

44        nX=cameraPosition[1];
45        nY=cameraPosition[5];
46        nZ=cameraPosition[9];
47        aux=sqrtf(nX*nX+nY*nY+nZ*nZ);
48        nX/=aux; nY/=aux; nZ/=aux;

50        posX=eyeX+vX*normV;
51        posY=eyeY+vY*normV;
52        posZ=eyeZ+vZ*normV;
53        stepX=vX;
54        stepY=vY;
55        stepZ=vZ;
56        hit=false;
57        for (int i=0; i<maxSteps; i++){
58          // remap position to grid coordinates
59          int x=posX+X/2;
60          int y=posY+Y/2;
61          int z=posZ+Z/2;

63          if(0<=x&&x<X && 0<=y&&y<Y && 0<=z&&z<Z){ //if in volume boundaries
64            volumeType sample=volume[X*Y*z+X*y+x]; //sample volume
65            if(sample>0){
66              i=maxSteps;
67              hit=true;
68            }
69            else{
70              posX += stepX;
71              posY += stepY;
72              posZ += stepZ;
73            }
74          }
75          else{
76            i=maxSteps;
77          }
78        }
79        depthMap[globalTidY*width+globalTidX]=hit?(depthMapType) sqrtf((posX-eyeX)*(
      posX-eyeX)+(posY-eyeY)*(posY-eyeY)+(posZ-eyeZ)*(posZ-eyeZ)):0;
80        //depthMap[globalTidY*width+globalTidX]=hit?(depthMapType) ((posX-eyeX)*nX+(
      posY-eyeY)*nY+(posZ-eyeZ)*nZ):0; // distance to projection plane.
81      }
82    }
83  }
```

LISTADO A.3: *Kernel* que genera, mediante trazado de rayos, un mapa de profundidad a partir de una representación del ambiente almacenada en un volumen.

Apéndice B

CUDA con Matlab

El aumento en el uso de la GPU como procesador auxiliar ha provocado que las aplicaciones matemáticas más usadas (Maple, MATLAB, Mathematica, etc.) provean interfaces para procesamiento en la GPU. En esta sección analizaremos cómo hacerlo con Matlab, puesto que varios grupos de trabajo utilizan este *software*. Para dar ejemplos, damos equivalencias con el ejemplo del capítulo segundo.

Como ya hemos visto, un kernel necesita entradas para poder trabajar. Generalmente, las entradas deben copiarse de memoria RAM del equipo anfitrión a la memoria RAM de la GPU. Hacer esto, en Matlab, es fácil con el comando `gpuArray()`. Por ejemplo,

```
dataGPU= gpuArray(dataCPU);
minPerBlockGPU=gpuArray(single(zeros(numberOfBlocks)));
minPerBlockIdxGPU =GpuArray(single(zeros(numberOfBlocks)));
```

copia el arreglo con los datos iniciales `dataCPU` del CPU al arreglo en GPU y reserva espacio para los dos arreglos de salida del *kernel* descrito en el capítulo segundo. Cuando se utilicen matrices, es importante conocer que, de manera predeterminada, las variables en Matlab son de doble precisión (`double`), así que es necesario hacer una conversión a `single` si se quiere trabajar con tipo `float` en el *kernel*. Otra nota importante es que las matrices, en Matlab, se almacenan por columnas y, por tanto, se transfiere al GPU como un arreglo unidimensional almacenado de esa manera.

Una vez que tenemos datos en memoria, necesitamos un *kernel* que trabaje sobre esos datos. Para esto creamos un objeto `CUDAKernel`.

```
k=parallel.gpu.CUDAKernel( ejemplo.ptx , ejemplo.cu );
```

El primer parámetro es el archivo .ptx (generable utilizando la bandera `-ptx` con el compilador, `nvcc -ptx miKernel.cu`) y el segundo es el archivo .cu. Si no se tiene el archivo .cu, se puede porporcionar, como una cadena, los tipos que recibe el *kernel* (por ejemplo, `const float*, float*, int*, int`).

El objeto `CUDAKernel` que generamos, con el *kernel* a ejecutar, tiene varias propiedades configurables, entre ellas, la configuración de lanzamiento.

```
 k.GridSize = [NB 1];
k.ThreadBlockSize = [TPB 1];
```

Después de configurar el *kernel* podemos ejecutarlo. Esto se realiza con la función: `[y1 y2 y3 ... yn]=feval(kernelHandle, input1, ..., inputn)`. Cada elemento `yi` apunta a los elementos de entrada `inputi` que no fueron declarados como constantes (`const`). Por ejemplo, para lanzar el kernel del ejemplo escribimos:

```
[minPerBlockGPU minPerBlockIdxGPU]=feval(k,dataGPU,minPerBlockGPU,
                                         minPerBlockIdxGPU,N+extra);
```

Y dado que el primer parámetro de entrada fue declarado como `const float*`, las variables de salida, `minPerBlockGPU` y `minPerBlockIdxGPU`, apuntan a los parámetros de entrada con esos nombres.

Cuando ejecutamos un *kernel* requerimos, usualmente, regresar el resultado a CPU para terminar el procesamiento o para dar una salida. Esto se puede hacer con la función `gather()`. Por ejemplo,

```
minPerBlockCPU=gather(minPerBlockGPU);
minPerBlockIdxCPU=gather(minPerBlockIdxGPU);
```

Existen algunos comandos útiles como `reset(gpuDevice)` y `gpuDevice()`. El primero, elimina todas las variables y limpia la memoria (utilizada por Matlab) de la GPU; y el segundo, despliega información de la tarjeta.

Como se observa, ejecutar un *kernel* con Matlab es muy sencillo. Al combinar todas las capacidades de esta herramienta con CUDA se pueden obtener resultados efectivos y en poco tiempo. Además de ejecutar *kernels*, se pueden utilizar comandos ya implementados, como la transformada rápida de Fourier, que utilizan la GPU. Este tipo de herramientas pueden ayudar a incursionar en el procesamiento con GPU. Se puede encontrar más información de la interfaz en [47].

Bibliograf´a

[1] NVIDIA. *CUDA C Programming Guide* [en línea], . http://docs.nvidia.com/cuda/cuda-c-programming-guide [Accedido: 20-II-2015].

[2] Kurt Oeler. *SGI to slash workstation prices* [en línea], 1998. http://news.cnet.com/SGI-to-slash-workstation-prices/2100-1001_3-213534.html [Accedido: 20-II-2015].

[3] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. *NVIDIA Tesla: A Uni ed Graphics and Computing Architecture. IEEE Micro*, 28(2):39 55, March 2008. ISSN 0272-1732. doi: 10.1109/MM.2008.31. URL http://dx.doi.org/10.1109/MM.2008.31.

[4] C.M. Wittenbrink, E. Kilgari , and A. Prabhu. *Fermi GF100 GPU Architecture. Micro, IEEE*, 31(2):50 59, March 2011. ISSN 0272-1732. doi: 10.1109/MM.2011.24.

[5] NVIDIA. *NVIDIA s Next Generation CUDA Compute Architecture: Fermi* [en línea], 2010. http://www.nvidia.com/content/pdf/fermi_white_papers/nvidiafermicomputearchitecturewhitepaper.pdf [Accedido: 3-III-2015].

[6] NVIDIA. *NVIDIA GeForce GTX 680. The fastest, most e cient GPU ever built.* [en línea], 2012. http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf [Accedido: 4-III-2015].

[7] NVIDIA. *NVIDIA s Next Generation CUDA Compute Architecture: Kepler* [en línea], 2013. http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf [Accedido: 4-III-2015].

[8] NVIDIA. *NVIDIA GeForce GTX 980 Featuring Maxwell, The Most Advanced GPU Ever Made.* [en línea], 2014. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF [Accedido: 5-III-2015].

[9] John Cheng, Max Grossman, and Ty McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014.

[10] Shane Cook. *CUDA Programming: A Developer s Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013. ISBN 9780124159334, 9780124159884.

[11] NVIDIA. *CUDA C Programming Guide. Mathematical Functions Appendix* [en línea], . http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#mathematical-functions-appendix [Accedido: 7-IV-2015].

[12] NVIDIA. *NVIDIA Occupancy Calculator* [en línea], . http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls [Accedido: 8-IV-2015].

[13] NVIDIA. *NVIDIA Visual Pro ler* [en línea], . https://developer.nvidia.com/nvidia-visual-profiler [Accedido: 9-IV-2015].

[14] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010. ISBN 0131387685, 9780131387683.

[15] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2012. ISBN 0124159923, 9780124159921.

[16] Nicholas Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013. ISBN 9780133261509.

[17] Microsoft. *Meet Kinect for Windows* [en línea]. https://dev.windows.com/kinect [Accedido: 13-I-2016].

[18] Je rey Meisner. *Collaboration, expertise produce enhanced sensing in Xbox One* [en línea]. http://blogs.microsoft.com/blog/2013/10/02/collaboration-expertise-produce-enhanced-sensing-in-xbox-one/ [Accedido: 20-V-2015].

[19] C.D. Mutto, P. Zanuttigh, and G.M. Cortelazzo. *Time-of-Flight Cameras and Microsoft Kinect*. SpringerBriefs in Electrical and Computer Engineering. Springer, 2012. ISBN 9781461438076.

[20] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, 47(1-3): 7 42, 2002. ISSN 0920-5691.

[21] Nicolas Burrus. *Kinect Calibration* [en línea], . http://nicolas.burrus.name/index.php/Research/KinectCalibration [Accedido: 2-II-2015].

[22] Je Kramer, Nicolas Burrus, Daniel Herrera C., Florian Echtler, and Matt Parker. *Hacking the Kinect*. Apress, Berkely, CA, USA, 1st edition, 2012. ISBN 1430238674, 9781430238676.

[23] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010. ISBN 1848829345, 9781848829343.

[24] Nicolas Burrus. *RGBDemo Software* [en línea], . http://rgbdemo.org/index.php [Accedido: 2-II-2015].

[25] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohli, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *IEEE ISMAR*. IEEE, October 2011. URL http://research.microsoft.com/apps/pubs/default.aspx?id=155378.

[26] Shahram Izadi, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, Dustin Freeman, Andrew Davison, and Andrew Fitzgibbon. Kinectfusion: Real-time 3d reconstruction and interaction using a moving depth camera. ACM Symposium on User Interface Software and Technology, October 2011. URL http://research.microsoft.com/apps/pubs/default.aspx?id=155416.

[27] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Computer Vision, 1998. Sixth International Conference on*, pages 839 846, Jan 1998. doi: 10.1109/ICCV.1998.710815.

[28] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH 96, pages 303 312, New York, NY, USA, 1996. ACM. ISBN 0-89791-746-4. doi: 10.1145/237170.237269. URL http://doi.acm.org/10.1145/237170.237269.

[29] Stanley Osher and Ronald P. Fedkiw. *Level set methods and dynamic implicit surfaces*. Applied mathematical science. Springer, New York, N.Y., 2003. ISBN 0-387-95482-1.

[30] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan. Interactive ray tracing for isosurface rendering. In *Visualization 98. Proceedings*, pages 233 238, Oct 1998. doi: 10.1109/VISUAL.1998.745713.

[31] Szymon Rusinkiewicz and Marc Levoy. E cient variants of the ICP algorithm. In *Third International Conference on 3D Digital Imaging and Modeling (3DIM)*, June 2001.

[32] Jiawen Chen, Dennis Bautembach, and Shahram Izadi. Scalable real-time volumetric surface reconstruction. *ACM Trans. Graph.*, 32(4):113:1 113:16, July 2013. ISSN 0730-0301. doi: 10.1145/2461912.2461940. URL http://doi.acm.org/10.1145/2461912.2461940.

[33] Matthias Nie ner, Michael Zollhofer, Shahram Izadi, and Marc Stamminger. Real-time 3d reconstruction at scale using voxel hashing. *ACM Trans. Graph.*, 32(6):169:1 169:11, November 2013. ISSN 0730-0301. doi: 10.1145/2508363.2508374. URL http://doi.acm.org/10.1145/2508363.2508374.

[34] A. Bapat, A. Ravi, and S. Raman. An iterative, non-local approach for restoring depth maps in rgb-d images. In *Communications (NCC), 2015 Twenty First National Conference on*, pages 1 6, Feb 2015. doi: 10.1109/NCC.2015.7084819.

[35] Yue Gao, You Yang, Yi Zhen, and Qionghai Dai. Depth error elimination for rgb-d cameras. *ACM Trans. Intell. Syst. Technol.*, 6(2):13:1 13:16, April 2015. ISSN 2157-6904. doi: 10.1145/2735959. URL http://doi.acm.org/10.1145/2735959.

[36] Nadia Figueroa, Haiwei Dong, and Abdulmotaleb El Saddik. A combined approach toward consistent reconstructions of indoor spaces based on 6d rgb-d odometry and kinectfusion. *ACM Trans. Intell. Syst. Technol.*, 6(2):14:1 14:10, March 2015. ISSN 2157-6904. doi: 10.1145/2629673. URL http://doi.acm.org/10.1145/2629673.

[37] Abel Pacheco. Adecuación de técnicas de descripción visual utilizando información 3d y su aplicación en robótica. Master s thesis, Universidad Nacional Autónoma de México, 2011.

[38] K. Wang, G. Zhang, and H. Bao. Robust 3d reconstruction with an rgb-d camera. *Image Processing, IEEE Transactions on*, 23(11):4893 4906, Nov 2014. ISSN 1057-7149. doi: 10.1109/TIP.2014.2352851.

[39] M. Keller, D. Lefloch, M. Lambers, S. Izadi, T. Weyrich, and A. Kolb. Real-time 3d reconstruction in dynamic scenes using point-based fusion. In *3D Vision - 3DV 2013, 2013 International Conference on*, pages 1 8, June 2013. doi: 10.1109/3DV.2013.9.

[40] Michael Zollhofer, Matthias Nie ner, Shahram Izadi, Christoph Rehmann, Christopher Zach, Matthew Fisher, Chenglei Wu, Andrew Fitzgibbon, Charles Loop, Christian Theobalt, and Marc Stamminger. Real-time non-rigid reconstruction using an rgb-d camera. *ACM Trans. Graph.*, 33(4):156:1 156:12, July 2014. ISSN 0730-0301. doi: 10.1145/2601097.2601165. URL http://doi.acm.org/10.1145/2601097.2601165.

[41] Richard A. Newcombe, Dieter Fox, and Steven M. Seitz. Dynamicfusion: Reconstruction and tracking of non-rigid scenes in real-time. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[42] Honda. *ASIMO* [en línea]. http://asimo.honda.com/ [Accedido: 2-I-2016].

[43] PCL. *Point Cloud Library (PCL)* [en línea]. http://pointclouds.org/ [Accedido: 22-VI-2015].

[44] Marsette Vona. *Moving Volume KinectFusion* [en línea]. www.ccs.neu.edu/research/gpc/rxkinfu/index.html [Accedido: 23-VI-2015].

[45] Niessner et. al. *Real-time 3D Reconstruction at Scale using Voxel Hashing* [en línea]. http://www.graphics.stanford.edu/ niessner/niessner2013hashing.html [Accedido: 23-VI-2015].

[46] Ling Shao, Jungong Han, Pushmeet Kohli, and Zhengyou Zhang. *Computer Vision and Machine Learning with RGB-D Sensors*. Springer Publishing Company, Incorporated, 2014. ISBN 3319086502, 9783319086507.

[47] Mathworks. *MATLAB GPU Computing Support for NVIDIA CUDA-Enabled GPUs* [en línea]. http://www.mathworks.com/discovery/matlab-gpu.html [Accedido: 29-VII-2015].