UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA EN COMPUTACIÓN

CÓMPUTO HETEROGÉNEO PARA PROCESAMIENTO DE NUBES DE PUNTOS MEDIANTE CPU, FPGA Y GPU Y SUS APLICACIONES EN ROBOTS MÓVILES

TESIS

QUE PARA OPTAR POR EL GRADO DE:

MAESTRO EN INGENIERÍA (COMPUTACIÓN)

PRESENTA:
RAMÓN NONATO LAGUNAS SÁNCHEZ

TUTOR:
JESÚS SAVAGE CARMONA
FACULTAD DE INGENIERÍA

MÉXICO, D.F. MAYO 2015

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Agradecimientos:

A mis padres: Ramón Lagunas y Teresa Sánchez, y mis hermanos: David y Claudia. Por todo su amor y por el esfuerzo realizado a lo largo de todos estos años para darme educación, salud y un hogar.

A mi esposa y amiga: María Soledad Calderón, por su amor y apoyo incondicionales, por su paciencia e inteligencia, y por ser la mujer que salió de mis sueños para convertirse en una realidad.

Al Dr. Jesús Savage Carmona, por darme la oportunidad de trabajar bajo su tutela y enseñarme que, el único obstáculo que existe para conseguir lo que uno desea se encuentra en la mente.

A todos mis amigos y profesores que directa o indirectamente intervinieron en la realización de este trabajo.

A CONACyT por el financiamiento otorgado durante mis estudios de posgrado.

Al Programa Universitario de Altera (Altera University Program) por la donación de la tarjeta de desarrollo DE2i-150 utilizada en este trabajo.

Se agradece a la DGAPA-UNAM por el apoyo proporcionado para la realización de esta tesis a través del proyecto PAPIIT IN117612, "Robot de Servicio par Asistencia a Adultos Mayores y en Sistemas Hospitalarios"

RESUMEN

En el área de robótica de servicio, se requiere que un robot ejecute acciones de manera autónoma con base en el entorno que lo rodea. Algunas tareas habituales que se presentan en esta área son: detección, reconocimiento y manipulación de objetos; navegación autónoma; detección y evasión de obstáculos; detección de personas y reconocimiento de voz; entre otras. Para lograr que un robot lleve a cabo estas tareas, es necesario que este pueda adquirir datos del entorno que lo rodea para, posteriormente, procesarlos y obtener información relevante. La adquisición de datos es posible realizarla mediante dispositivos conocidos como sensores, mientras que el procesamiento de estos datos puede ser llevado a cabo implementando distintos algoritmos en una computadora. Dicha computadora debe ser capaz de soportar múltiples procesos ejecutándose de manera concurrente, idealmente un proceso por cada componente del robot. Esto provoca que entre más componentes conformen al robot, se requiera más poder de cómputo para obtener un rendimiento óptimo. Este incremento en el poder de cómputo es posible lograrlo aumentando las capacidades de la computadora central (de propósito general) del robot o bien, distribuyendo los múltiples procesos haciendo uso de múltiples Unidades de Procesamiento (UP) de propósito específico. En las últimas décadas, el rápido crecimiento tecnológico ha permitido que hoy en día se cuenten con UP comerciales de propósito específico con grandes capacidades de procesamiento. En el presente trabajo se realiza un estudio comparativo entre el procesamiento de datos realizado en una computadora central y el procesamiento realizado en sistemas de cómputo heterogéneos (combinando computadoras de propósito general y UP de propósito específico). Para esto, se aborda la tarea de detección de objetos haciendo uso de imágenes 3D; se hace uso de un CPU Multicore (como computadora central) y de GPU y diseños en FPGA (como UPs de propósito específico). Se busca comparar el desempeño de los distintos esquemas de procesamiento y mostrar que los sistemas de cómputo heterogéneo ofrecen mejor rendimiento que los sistemas centralizados.

INDICE DE CONTENIDO

Capítulo 1.Introducción...9
1.1.Objetivos...12
1.2.Motivación...13
1.3.Organización de la tesis...14
Capítulo 2.Marco teórico...15
2.1.Arquitecturas y sistemas de hardware...16
2.1.1.CPU...17
2.1.2.FPGA...20
2.1.3.GPU...24
2.1.4.Sistemas Embebidos y Cómputo Heterogéneo...27
2.2.Cámaras RGB-D...29
Capítulo 3.Descripción del problema...31
3.1.Planteamiento...31
3.2.Extracción de puntos característicos...33
3.3.Segmentación de la superficie plana dominante...35
Capítulo 4.Esquema de procesamiento en CPU...39
4.1.Diagramas de clases...40
4.1.1.Extracción de puntos característicos...40
4.1.2.Segmentación de la superficie plana dominante...41
4.2.Diagramas de flujo...44
4.2.1.Extracción de puntos característicos...44
4.2.2.Segmentación de la superficie plana dominante...47
Capítulo 5.Esquema de procesamiento en FPGA...51
5.1.Sistema de memoria...53
5.2.Módulo de extracción de puntos característicos...56
5.2.1.Memoria de nube de puntos y memoria de vectores normales...56
5.2.2.Registros A, B, fila y col...57
5.2.3.Módulo producto-cruz...57
5.2.4.Módulo generador fila-columna...59
5.2.5.Unidad de control...60

Capítulo 6.Esquema de procesamiento en GPU....63
6.1.Arquitectura de hardware heterogéneo (CPU+GPU)....64
6.1.1.Configuración básica CPU/GPU....64
6.1.2.Configuración CPU/GPU con múltiples CPU....65
6.1.3.Configuración CPU/GPU con múltiples GPU....66
6.2.Arquitectura de software CUDA....69
6.2.1.Estructura de un programa con CUDA C....69
6.3.Extracción de puntos característicos....72
6.4.Segmentación de la superficie plana dominante....75
Capítulo 7.Experimentos y resultados....79
7.1.Comparación del tiempo de ejecución....79
7.1.1.Extracción de puntos característicos....79
7.1.2.Segmentación de la superficie plana dominante....81
7.2.Comparación de la precisión de datos....82
7.2.1.Extracción de puntos característicos....82
7.3.Comparación de la precisión del proceso....83
7.3.1.Extracción de puntos característicos....83
Capítulo 8.Conclusiones y trabajo futuro....87
Apéndice A.Sistema de transmisión y recepción de datos entre CPU y FPGA....91
A.1 Tarjeta de desarrollo Intel De2i-150....91
A.1.1. El procesador Intel Atom N2600....92
A.1.2. El FPGA Altera Cyclone IV EP4CGX150DF31....93
A.2 Sistema de hardware-software para transferencia de datos....93
A.2.1. Controlador de software....93
A.2.2. Interfaz de hardware....94
A.2.3. Sistema de sincronización hardware-software....95
Índice de figuras....99
Índice de tablas....101
Bibliografía....103

Capítulo 1. Introducción

En diversas industrias: médica, aeronáutica, comunicaciones y robótica, (por mencionar algunas) solucionar un problema implica la implementación de distintas aplicaciones (cada una ejecutándose en una plataforma distinta e intercambiando datos entre sí) encapsuladas en un único sistema. Para lograr esto, se necesitan conocer a fondo los algoritmos y las distintas plataformas a utilizar, así como también, establecer una serie de políticas para lograr que la comunicación entre plataformas se realice de manera eficiente. Al estudio y desarrollo de este tipo de sistemas (conformados por múltiples plataformas) se denomina Cómputo Heterogéneo.

En cómputo heterogéneo, un sistema hace uso de más de un tipo de procesador para llevar a cabo una tarea, es decir, son sistemas multiprocesador. Además, estos sistemas, cuentan con una red de comunicación que interconecta a todos los procesadores ([LD09]) con el objetivo de intercambiar datos entre ellos. Para obtener el máximo rendimiento de este tipo de sistemas, es importante que los procesadores cuenten con capacidades de procesamiento diferentes (procesadores especializados a un tipo de aplicación en particular) y que estas capacidades se complementen entre sí. Lo anterior, implica hacer uso de distintas arquitecturas de hardware (cada una con un conjunto de instrucciones propio) interactuando entre sí. Es importante que esta interacción se realice de manera robusta, dadas las distintas incompatibilidades que puedan presentar las arquitecturas involucradas en un sistema de cómputo heterogéneo.

Actualmente, existen diversos sistemas que hacen uso del cómputo heterogéneo para la resolución de tareas. Un ejemplo de estos sistemas son los micro-chips que integran un procesador de propósito general (CPU) con una unidad de procesamiento gráfico (GPU). Esta configuración permite obtener lo mejor de ambos mundos: mientras el GPU se encarga de las tareas intensivas de procesamiento gráfico, el CPU ejecuta los procesos del sistema operativo. La combinación (CPU+GPU) ha hecho posible obtener mejoras en el rendimiento de las computadoras personales. Además, en años recientes, los GPU han sido altamente usados para el desarrollo de aplicaciones de propósito general, gracias a la posibilidad de ejecutar código en ellos (programado mediante un lenguaje de alto nivel) y al alto poder de cómputo que ofrecen.

Al desarrollar una aplicación para alguna arquitectura específica (CPU, GPU o cualquier otra), se debe tomar en cuenta que tal aplicación respete ciertos patrones de diseño con el fin de obtener el máximo rendimiento de dichas arquitecturas. Así mismo, existen aplicaciones que, por naturaleza propia, presentan el mismo desempeño al ser ejecutadas en distintas arquitecturas. Para lograr una mejoraría en el rendimiento de este tipo de aplicaciones es necesario, mas allá de optimizar la implementación en software, crear nuevas arquitecturas de hardware, de propósito específico, que respondan a la demanda de dichas aplicaciones.

Comparado con el proceso de creación de software, la fabricación de sistemas de hardware suele ser un proceso costoso en muchos aspectos. A pesar de eso, y como menciona [Som04], los sistemas con diseños propios en Hardware ofrecen un mejor desempeño a costa de un desarrollo tardado y poca flexibilidad en cuanto a cambios. En las últimas 4 décadas, las desventajas, en cuanto a costos, del desarrollo de hardware han visto una significativa reducción, gracias a la aparición de dispositivos que permiten *programar* hardware en su interior. A estos dispositivos se les conoce como Dispositivos Lógicos Programables (PLD).

Los PLD son componentes electrónicos, implementados en un chip, que no cumplen con una función específica, es decir, la funcionalidad de un PLD puede ser diseñada en base a las necesidades del diseñador de hardware. Para lograr que un PLD cumpla con una funcionalidad requerida, el diseñador deberá *crear* dicha funcionalidad (a este paso se le conoce como descripción de hardware) mediante un lenguaje especializado conocido como Lenguaje de Descripción de Hardware (HDL). Los HDL son lenguajes de programación de alto nivel con una sintaxis especializada para definir la estructura, diseño y operación de circuitos electrónicos digitales [Bha95]. El uso de los HDL en conjunto con los PLD permite construir sistemas de hardware con menor costo y en menor tiempo (en comparación al requerido para construir un chip de propósito específico). Además, los sistemas diseñados en un PLD, ofrecen una alta flexibilidad en cuanto a cambios, dado que el PLD puede configurarse mas de una vez y cada configuración implica solamente cambios en el código HDL.

Gracias a los avances en la fabricación de hardware y desarrollo de software es posible, hoy en día, combinar lo mejor de ambos mundos (hardware y software) en un sólo sistema para así satisfacer las demandas actuales en cuanto a tiempos de respuesta en aplicaciones críticas. Aplicaciones como las que se presentan en la industria de la robótica, la cual ha tenido un crecimiento notable en épocas recientes.

Desde la introducción de los robots a la industria en los 70s poco a poco la robótica ha ganado popularidad y ha tenido un incremento considerable en los últimos años. Sin embargo, como menciona *Bill Gates* en su artículo *"A Robot in Every Home"* (véase [Gates07]) existen distintos problemas en robótica aun sin resolver, tales como son Aprendizaje Máquina, Navegación y Visión por Computadora. Por este mismo motivo, en el año de 1997, surgió la Competencia Internacional de Robótica llamada RoboCup, la cual tiene como objetivos principales la resolución de estos problemas (Aprendizaje, Navegación y Visión) e impulsar la salida de los robots de la industria hacia sectores como deportes, oficinas y hogares. La integración de los robots a este *nuevo mundo*, implica una fuerte interacción con los seres humanos y con objetos de uso cotidiano (muebles, comida, herramientas, etc.); tal interacción, debe realizarse de tal forma que se garantice la seguridad física de los seres humanos.

De acuerdo con [NIAM04], la seguridad física de los humanos puede lograrse dotando al robot de algoritmos de evasión de obstáculos (personas y objetos) mediante el uso de dispositivos que le permitan al robot percibir el ambiente que lo rodea. Algunos de estos dispositivos (también conocidos como sensores) pueden ser: cámaras, sensores ultrasónicos, sensores infrarrojos, sensores láser, etc. Uno de los sensores más populares utilizados en robótica son las cámaras RGB-D. Esto no es de sorprenderse dado que, de todos los sistemas de percepción del ser humano, el sistema de visión es el más importante y este tipo de cámaras permiten obtener información en 2 y 3 dimensiones. Por este motivo, en robótica, el campo de visión por computadora ha sido uno de los más explorados en los últimos años.

Las aplicaciones desarrolladas en el área de visión por computadora demandan tiempos de respuesta relativamente cortos. Sin embargo, la información percibida por los sensores RGB-D suele contener un gran volumen de datos y, en ocasiones, procesar esos datos se vuelve un cuello de botella en distintas aplicaciones. Es en este punto en el que surge la necesidad de desarrollar aplicaciones de visión por computadora haciendo uso de técnicas de cómputo heterogéneo, con el fin de determinar la viabilidad de este tipo de sistemas en campos críticos como la robótica.

En el presente trabajo se realiza un análisis del desempeño de distintos algoritmos de visión por computadora, enfocados al procesamiento de datos 3D, implementados en sistemas de cómputo heterogéneo. Para llevar a cabo esto, fueron elegidas tres distintas arquitecturas de hardware: CPU, GPU y un tipo de PLD conocido como FPGA (del inglés *Field Programable Gate Array*).

1.1. Objetivos

Este trabajo presenta un estudio comparativo del desempeño de tres sistemas de cómputo heterogéneo (CPU+FPGA, CPU+GPU y CPU doble núcleo) mediante la implementación de algoritmos de visión por computadora (orientados al procesamiento 3D) en cada uno de ellos. Los objetivos generales son:

- Conocer el estado del arte en arquitecturas de hardware.
- Conocer el estado del arte en procesamiento de datos 3D.
- Analizar el desempeño de sistemas embebidos en procesamiento de datos 3D.

Los objetivos particulares que se derivan son:

- Tener acceso a investigaciones actuales como fuentes originales de información.
- Implementar algoritmos de procesamiento 3D en un procesador de propósito general.
- Implementar algoritmos de procesamiento 3D en un sistema de cómputo heterogéneo (CPU-FPGA).
- Implementar algoritmos de procesamiento 3D en un sistema de cómputo heterogéneo (CPU-GPU).
- Aplicar distintos algoritmos de procesamiento 3D a un proceso de robótica móvil.

1.2. Motivación

En años recientes, el desarrollo de algoritmos de procesamiento de datos 3D (o procesamiento de nubes de puntos) ha ido en aumento. Lo anterior es debido a la aparición de nuevas tecnologías empleadas para la adquisición de datos 3D y a los nuevos problemas que día con día se plantean en esta área. Es preciso dar solución a estos problemas y aplicarlos en procesos de la vida cotidiana.

En 2010 comenzó el desarrollo de un proyecto de software libre a nivel mundial llamado PCL (Point Cloud Library, [PCL15]). El proyecto PCL, dese sus inicios hasta la fecha, pretende dar solución a distintos problemas planteados en el área de procesamiento de nubes de puntos como: reconstrucción de superficies, segmentación, extracción de características, etc. En el desarrollo de esta librería se encuentran involucradas más de 60 universidades y compañías a nivel mundial, entre ellas: Intel, nVIDIA, Eindhoven University of Technology, University of Toronto, entre otras, [BC11]. En la actualidad ninguna institución mexicana se encuentra participando activamente en el desarrollo de esta librería.

Hoy en día, en el laboratorio de Bio-Robótica de la UNAM, se hace uso de distintas técnicas de procesamiento de nubes de puntos para dar solución a problemas en el área de robótica de servicio. Algunos problemas pertenecientes a esta área son: detección de objetos situados sobre superficies planas, detección de obstáculos en la trayectoria de un robot móvil, detección de gestos, entre otros. La solución de estos problemas específicos significan un paso más hacia la fabricación de robots móviles de propósito general.

Así mismo, en el laboratorio de Bio-Robótica, se desarrolla el proyecto *Justina* (Figura 1.1), que tiene como objetivo principal la construcción de un robot móvil de servicio para la asistencia a personas en tareas generales del hogar. Este robot, cuenta con sistemas de procesamiento de audio, visión computacional, navegación, inteligencia artificial y control; y ha sido el resultado de más de 15 años de investigación en robótica bajo el mando del Dr. Jesús Savage Carmona.

La implementación de soluciones, para el procesamiento de nubes de puntos, en el robot *Justina* se vuelve una tarea prioritaria hacia la consecución del objetivo principal de dicha investigación. El presente trabajo de tesis pretende aportar desarrollo e investigación tecnológica en las áreas de sistemas embebidos, procesamiento de nubes de puntos y cómputo heterogéneo; con la finalidad de solucionar problemas pertenecientes al área de robótica de servicio relacionados con el procesamiento de nubes de puntos. Así mismo se busca medir el desempeño de distintos sistemas de cómputo aplicados a un problema en particular.

1.3. Organización de la tesis

Capítulo 2. Marco teórico. En este capítulo se presentan las bases teóricas y los trabajos existentes relacionados con esta investigación.

Capítulo 3. Descripción del problema. Descripción del problema. En este capítulo se presenta el planteamiento del problema a abordar: detección de objetos en escenas 3D.

Capítulo 4. Esquema de procesamiento en CPU. En este capítulo se detalla la metodología seguida para la implementación de algoritmos de procesamiento 3D en el sistema CPU Multicore.

Capítulo 5. Esquema de procesamiento en FPGA. En este capítulo se detallan los módulos de hardware desarrollados para la implementación de algoritmos de procesamiento 3D en el FPGA.

Capítulo 6. Esquema de procesamiento en GPU. En este capítulo detalla la metodología seguida para la adaptación de los algoritmos del capítulo 4 a un esquema de procesamiento en paralelo.

Capítulo 7. Experimentos y resultados. En este capítulo se presentan los resultados de las pruebas de desempeño realizadas con los distintos sistemas de cómputo heterogéneo.

Capítulo 8. Conclusiones y trabajo futuro. En este capítulo se desarrollan las conclusiones obtenidas de los distintos experimentos realizados.

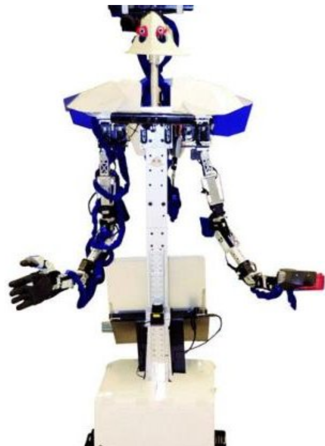

Figura 1.1: Robot de servicio Justina, proyecto desarrollado en el laboratorio de BioRobótica de la UNAM.

Capítulo 2. Marco teórico

Con la aparición de las cámaras RGB-D, las cuales permiten capturar información 3D del entorno, la investigación y desarrollo tecnológico en esta área experimentó un crecimiento notable. Debido a sus características (descritas a detalle más adelante en este capítulo) las cámaras RGB-D se han convertido en una opción viable para su uso en aplicaciones de procesamiento 3D, robótica, interfaces inteligentes, entre otras.

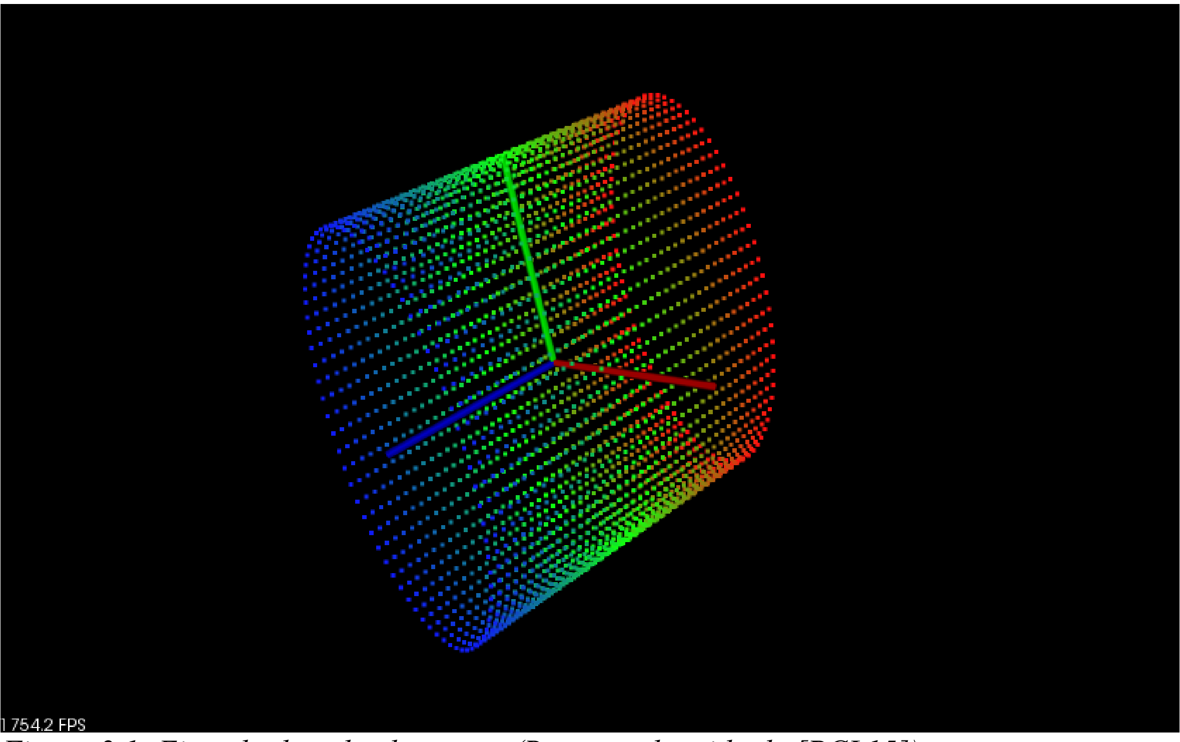

Figura 2.1: Ejemplo de nube de puntos.(Recurso obtenido de [PCL15])

La información 3D captada por estas cámaras es adquirida por la computadora en forma de *Nubes de Puntos*. Una nube de puntos es un conjunto de vértices en un sistema de coordenadas tridimensional. Estos vértices se identifican como coordenadas X, Y y Z y son representaciones de la superficie externa de un objeto (Figura 2.1). Una vez adquiridas desde una cámara RGB-D, las nubes de puntos pueden ser procesadas por medio de diferentes *Sistemas de Hardware*[1] y software. Este trabajo hace uso de tres distintos sistemas de hardware para el procesamiento de nubes de puntos.

1 Un *Sistema de Hardware* puede entenderse como un conjunto de componentes electrónicos interconectados que ejecutan un proceso específico. Los *Sistemas de Hardware Embebidos* y los *Sistemas de Hardware Heterogeneo* (descritos a detalle más adelante en esta capítulo) son sólo algunos ejemplos.

En este capítulo se presenta una breve introducción de los conceptos relacionados con procesamiento de nubes de puntos (algoritmos y bibliotecas de funciones), así mismo se describen las arquitecturas de hardware utilizadas durante el desarrollo de este trabajo. También se describe el funcionamiento de las cámaras RGB-D y las posibilidades que ellas han abierto en el desarrollo del campo. De manera particular se abordan los problemas de reducción de ruido en una nube de puntos, segmentación de planos y detección de objetos, y se describen distintos algoritmos que dan solución a estos problemas.

2.1. Arquitecturas y sistemas de hardware

Una *Arquitectura de Hardware* puede ser definida como la descripción o representación de los componentes electrónicos, y sus interrelaciones, de un sistema de hardware. En este apartado se detallan las distintas arquitecturas de hardware utilizadas durante el desarrollo de esta tesis.

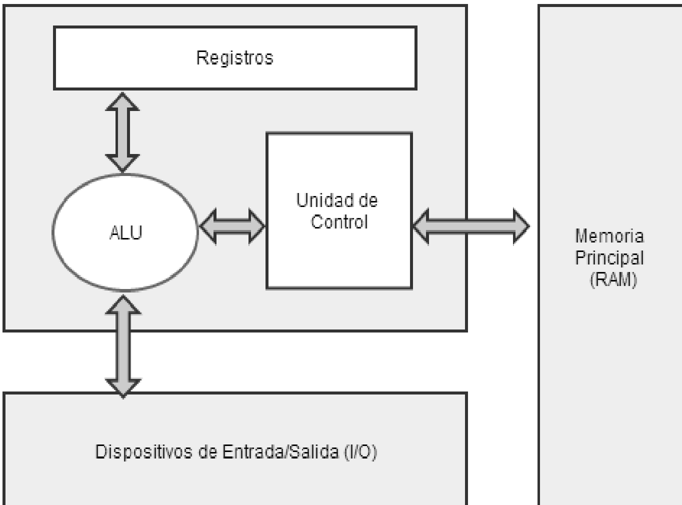

Figura 2.2: Componentes principales y estructura interna de un CPU. (Recurso obtenido de [SVC10])

2.1.1. CPU

La Unidad Central de Proceso (CPU, *Central Processing Unit*) o procesador es un componente electrónico que controla la operación de una computadora y lleva a cabo las operaciones de procesamiento de datos, [Sta97]. Para que un procesador pueda realizar dicho procesamiento de datos, este debe ejecutar una serie de instrucciones (instrucciones aritméticas y lógicas). Las instrucciones que un procesador puede ejecutar están definidas en el *Conjunto de Instrucciones* del procesador. En la Figura 2.2 se muestra la estructura interna y los componentes de un procesador. [Sta05] define los componentes de un procesador como:

- **Unidad de control**: Controla el funcionamiento del CPU.
- **Unidad Aritmético Lógica (ALU, *Arithmetic Logic Unit*)**: Lleva a cabo las funciones de procesamiento de datos del CPU.
- **Registros**: Proporcionan almacenamiento interno a la CPU.

Otra característica importante de un procesador es la *Frecuencia de Reloj,* que se refiere a la velocidad de oscilación del reloj interno del procesador y se encuentra medida en hertz. En los procesadores actuales, la Frecuencia de Reloj ronda los 3.5 GHz (en la Figura 2.3 se puede observar la evolución de la Frecuencia de Reloj en las computadoras a través de los años). La Frecuencia de Reloj es una medida importante dado que el desempeño de un procesador puede ser estimado en función de la Frecuencia de Reloj (como menciona [PH08]).

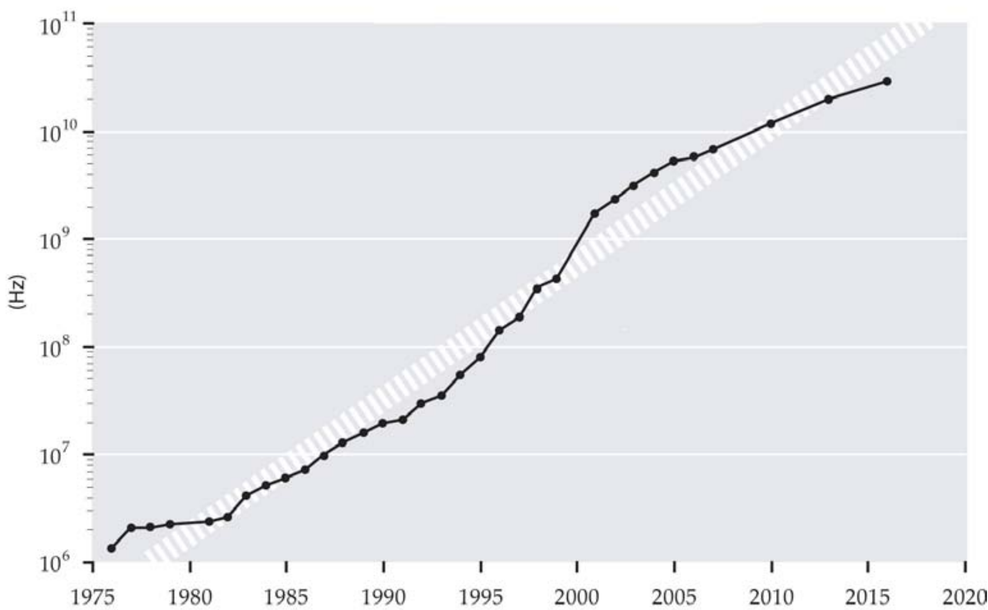

Figura 2.3: Evolución de la Frecuencia de Reloj de los procesadores desde 1975.

Un procesador, es únicamente un componente de un sistema de mayor jerarquía llamado Computadora. Una computadora es un dispositivo electrónico capaz de realizar cálculos numéricos obedeciendo instrucciones muy específicas y elementales que reflejan su estructura funcional y organizacional, [SVC10]. Existen diversos criterios de clasificación para las computadoras. Uno de ellos es la manera en cómo ejecutan sus cálculos, el cual origina dos categorías: aquellas que realizan los cálculos de manera secuencial y aquellas que realizan procesos en paralelo.

Las primeras constan de un único procesador ejecutando una instrucción a la vez, además, en este tipo de computadoras sólo se puede buscar o almacenar un elemento de datos a la vez. Para ejecutar una instrucción, este tipo de computadoras tienen que efectuar las siguientes etapas:

- Obtener el código de la instrucción a ejecutar.
- Decodificar la instrucción, es decir, saber cuales son las micro-operaciones que tiene que realizar la computadora para ejecutar dicha instrucción.
- Ejecutar la instrucción.
- Almacenar el resultado.

Estas etapas se ejecutan de manera secuencial para cada instrucción, por lo tanto, una nueva instrucción no puede comenzar a ejecutarse hasta que la anterior termine. La Figura 2.4 muestra un ejemplo de este tipo de computadora.

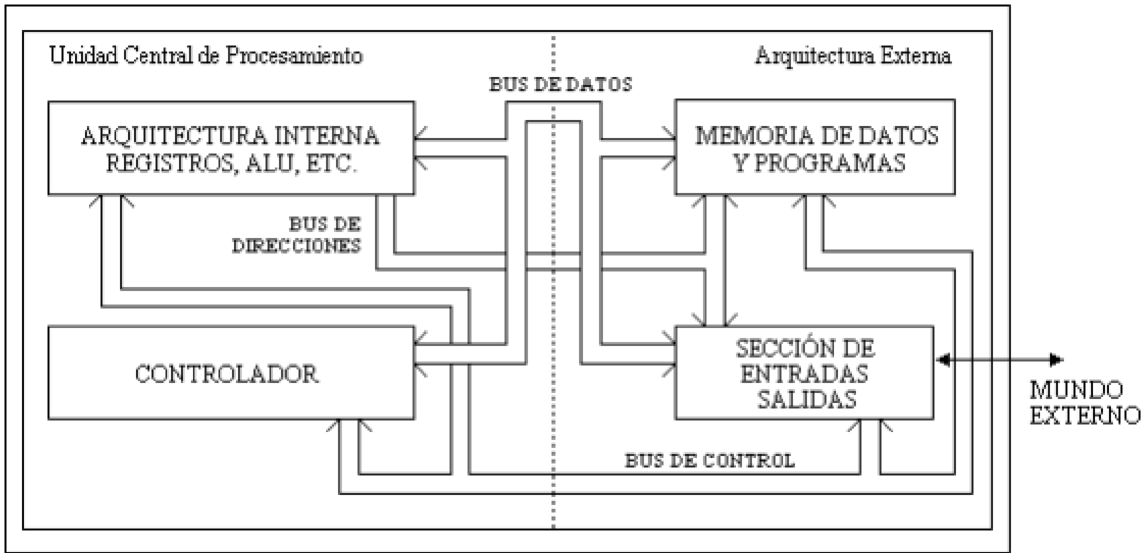

Figura 2.4: Computadora de von Neuman.(Recurso obtenido de [SVC10])

Las computadoras pertenecientes a la segunda clasificación cuentan con varios de sus componentes internos repetidos, de esta forma pueden ejecutar múltiples instrucciones al mismo tiempo. Dentro de esta clasificación de computadoras se encuentran las arquitecturas SIMD (*Single Instruction Multiple Data*) y MIMD (*Multiple Instruction Multiple Data*).

Las arquitecturas SIMD ejecutan la misma instrucción sobre múltiples datos simultáneamente. La principal característica de estas arquitecturas es que cuentan con una sola unidad de control y multiples procesadores. La Figura 2.5 muestra un modelo de arquitectura SIMD de tres procesadores.

Las computadoras con arquitectura MIMD son computadoras asíncronas con control descentralizado de hardware, es decir, cada procesador tiene su propia unidad de control ejecutando un programa diferente. Generalmente, las arquitecturas de este tipo consisten de varios procesadores autónomos que pueden ejecutar flujos independientes de instrucciones usando datos locales. La Figura 2.6 muestra un modelo de arquitectura MIMD.

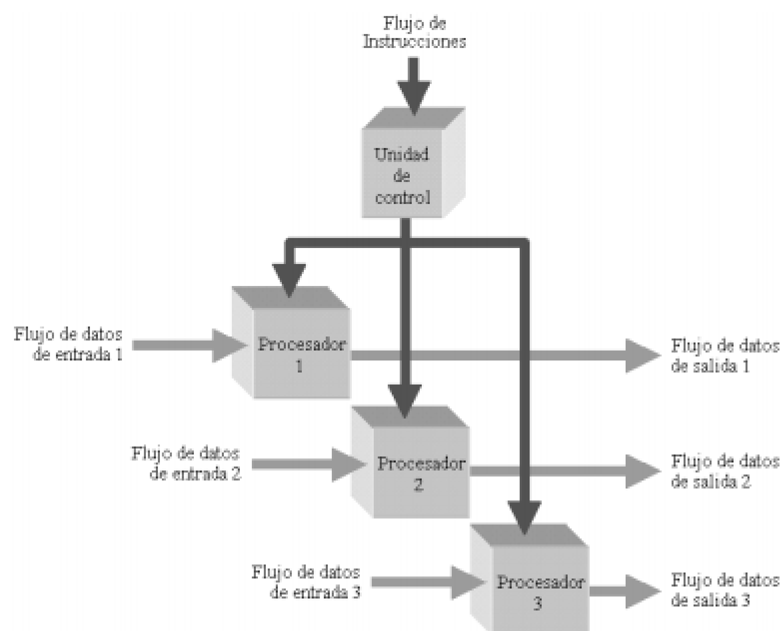

Figura 2.5: Arquitectura SIMD. (Recurso obtenido de [SVC10])

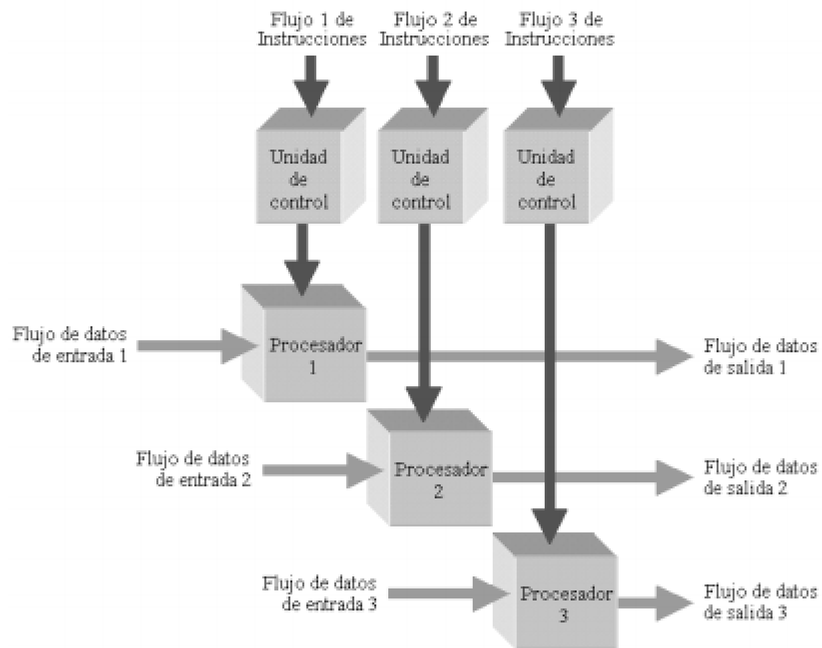

Figura 2.6: Arquitectura MIMD. (Recurso obtenido de [SVC10])

2.1.2. FPGA

Un FPGA (Arreglo de Compuertas de Campo Programables, *Field Programmable Gate Array*) es un dispositivo semiconductor programable que contiene bloques de lógica cuya interconexión y funcionalidad puede ser configurada mediante un lenguaje de programación especializado [X14]. La Figura 2.8 muestra una interpretación en alto nivel de un FPGA.

Cada FPGA está conformado por tres componentes principales: un arreglo de bloques lógicos (CLB, *Configurable Logic Block*), bloques de Entrada/Salida e interconexiones (las cuales conectan los CLBs entre ellos y con los bloques de Entrada/Salida). Los CLB son la unidad lógica básica de un FPGA, cada CLB está formado por un matriz de elementos lógicos (MUX, Flip-Flop, Sumador) con 4 o 6 canales de entrada de datos los cuales se encuentran conectados a dispositivos lógicos llamados LUT (un LUT representa una función booleana modelada con una tabla de verdad). Dicha matriz es altamente flexible y puede ser configurada para *crear* lógica combinatoria, registros o memorias RAM. En la Figura 2.8 se puede observar un diagrama de bloques que representa la estructura interna de un CLB.

Los bloques de Entrada/Salida proveen una interfaz entre el FPGA y el mundo exterior. Estos bloques pueden soportar diferentes estándares de Entrada/Salida como: Ethernet, VGA, PCI Express, USB, entre otros, y permiten comunicar el hardware descrito en un FPGA con dispositivos externos como cámaras, micrófonos, pantallas, etc.

Las interconexiones de un FPGA, al igual que los CLB, son altamente flexibles y pueden ser trazadas con base en las necesidades del diseñador de hardware, permitiendo así el envío de señales entre los CLB y los bloques de Entrada/Salida. Comúnmente el software de diseño de hardware para FPGAs se encarga de realizar las interconexiones, salvo algunos casos donde el diseñador requiera realizar las interconexiones de forma manual, esto reduce significativamente la complejidad del diseño de hardware.

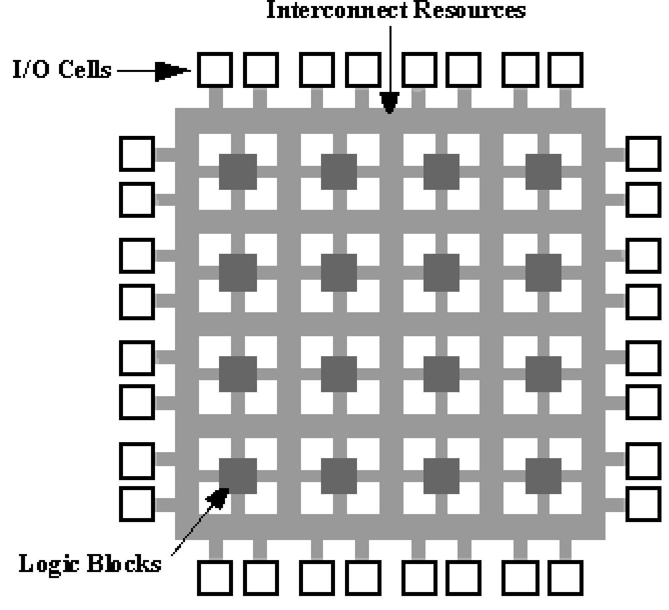

Figura 2.7: Diagrama de alto nivel que representa la estructura interna de un FPGA. Se pueden observar los distintos componentes del FPGA: interconexiones (Interconnect Resources), bloques de Entrada/Salida (I/O Cellls) y CLB (Logic Blocks).Recurso obtenido de [Free09].

Como ya se mencionó anteriormente, la funcionalidad de un FPGA puede ser configurada con base en las necesidades de cada diseñador de hardware. Para definir la funcionalidad de un FPGA se debe elaborar un *programa* en un lenguaje de programación conocido como HDL (Lenguaje de Descripción de Hardware, *Hardware Description Language*). Los HDL son lenguajes de programación de alto nivel con una sintaxis especializada para definir la estructura, diseño y operación de circuitos electrónicos digitales [Bha95]. Los HDL más populares actualmente son Verilog y VHDL.

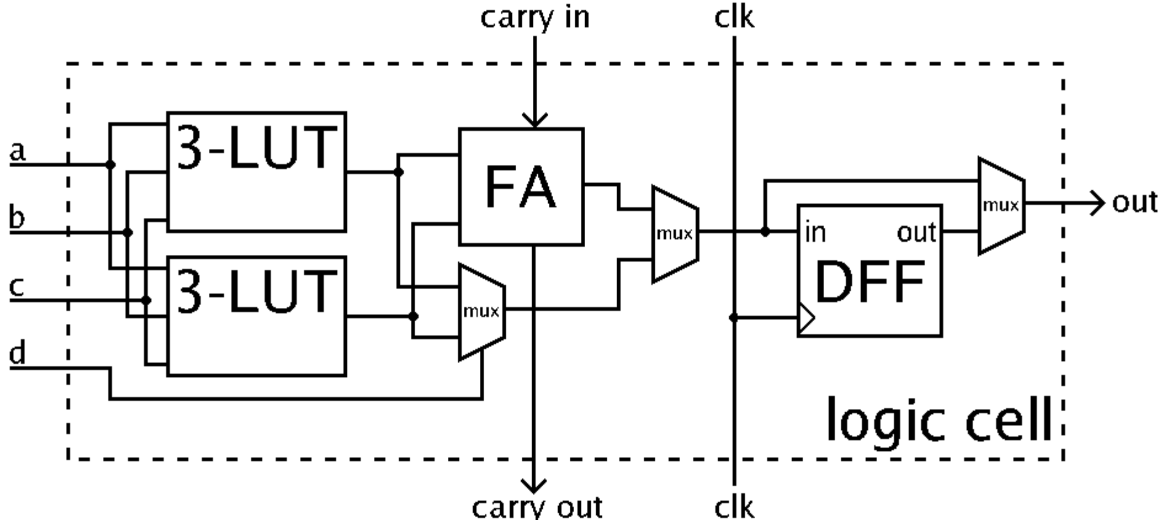

Figura 2.8: Diagrama de bloques de un CLB, contiene 2 LUT, 1 Sumador (FA) y un Flip Flop (FF). Recurso obtenido de [Kall10].

El diseño de circuitos digitales en un FPGA es un proceso metódico que involucra más de una etapa. En la Figura 2.9 se muestra el flujo de diseño de hardware digital para un FPGA y a continuación se detallan cada una de las etapas que lo conforman:

- **Entrada de diseño**: Se debe elegir el HDL con el cual se diseñará el hardware, posteriormente se realiza el diseño de hardware (programación con un HDL).
- **Síntesis de diseño**: La síntesis del diseño recoge toda la información de los archivos de entrada del diseño y los agrupa en un listado de las compuertas lógicas (*netlists*) del sistema y las conexiones entre éstas.
- **Implementación del diseño**: La implementación del diseño es un proceso que consiste en convertir las *netlist* en un archivo de configuración del dispositivo (*bitstream*) que consiste en una serie de bits que configuran las conexiones y los CLB del FPGA.

- **Programación del dispositivo**: Finalmente, el *bitstream* es descargado al FPGA para que este funcione de acuerdo al diseño.
- **Simulación funcional**: La simulación funcional consiste en comprobar que el funcionamiento del sistema es el indicado. En esta simulación se asignan valores a las entradas del sistema, y se analizan las salidas, comparándolas con las salidas esperadas. Cualquier divergencia entre éstas implicará la modificación del diseño.
- **Análisis de tiempos**: El análisis de tiempos permite comprobar el cumplimiento de las especificaciones de tiempo del sistema. Este paso es importante debido a que, una vez en el FPGA, las compuertas lógicas presentan retardos que deben ser tenidos en cuenta ya que pueden afectar al comportamiento general del sistema.
- **Verificación del dispositivo**: Una vez programado el FPGA, hay que comprobar que el sistema funciona según las especificaciones y los resultados de las diversas simulaciones. Para lograr esto es necesario extraer las señales internas del FPGA para poder analizarlas. Algunas herramientas para extraer estas señales pueden ser LEDs, displays, osciloscopios, entre otros.

Figura 2.9: Diagrama del flujo de diseño de un FPGA. Recurso obtenido de [GENERA15].

2.1.3. GPU

La Unidad de Procesamiento Gráfico (GPU, *Graphics Processing Unit*) es un circuito electrónico (multi-procesador) diseñado para acelerar el procesamiento de tareas gráficas en una computadora. Su arquitectura interna es de tipo SIMD (vista en el apartado 2.1.1) lo cual hace a las GPU dispositivos eficientes para la manipulación de grandes cantidades de datos.

A diferencia de un CPU (el cuál es un único procesador) una GPU consta de múltiples procesadores (llamados núcleos) los cuales comparten la misma arquitectura interna (ALU, memoria, unidad de control, registros, unidad de coma flotante). En la Figura 2.10 se muestra la comparativa, en alto nivel, entre un CPU y una GPU, como se puede observar la GPU contiene múltiples procesadores (núcleos) y cada uno de estos núcleos contienen su propia memoria cache para el almacenamiento de datos de procesamiento locales, así mismo, todos los núcleos comparten una misma memoria RAM.

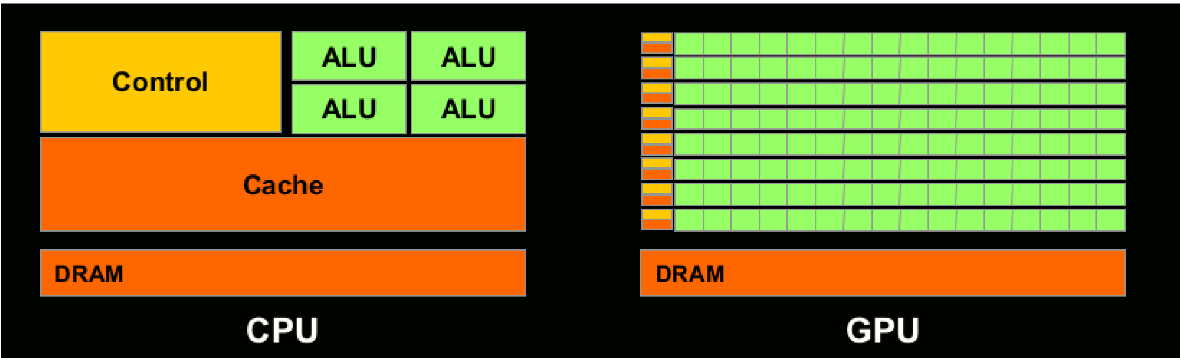

Figura 2.10: Comparativa entre las arquitectura interna de un CPU y una GPU. (Recurso obtenido de [NVIDIA15])

Una GPU se encuentra organizada de la siguiente manera:

- Cada GPU cuenta con diversos procesadores llamado **núcleos**, estos procesadores cuentan, cada uno, con: memoria cache, una ALU, una Unidad de Punto Flotante (FPU) y una unidad de control.
- Los núcleos se encuentran organizados en Multiprocesadores de Flujo (*Streaming Multiprocessor,* Figura 2.11), son procesadores que contienen múltiples núcleos. Cada *Streaming Multriprocessor* contiene aproximadamente 32 **núcleos** y un bloque de memoria (llamado *Memoria Compartida* debido a que todos los núcleos de este procesador la comparten) [NVIDIA09].

- Finalmente, los *Streaming Multiprocessors*, además de contar cada uno con una memoria local, comparten una Memoria Global llamada Memoria del Dispositivo (que es la memoria en donde comúnmente se almacenan imágenes o cualquier información que se desee procesar) Figura 2.12.

Debido al alto nivel de paralelismo en las GPU actuales, estas son usadas por la comunidad de desarrolladores no sólo para aplicaciones gráficas, sino también para aplicaciones científicas donde se manejan grandes volúmenes de datos de punto flotante (de acuerdo con [BDHHS10] es en este tipo de aplicaciones donde se obtienen el mejor rendimiento por parte de las GPU).

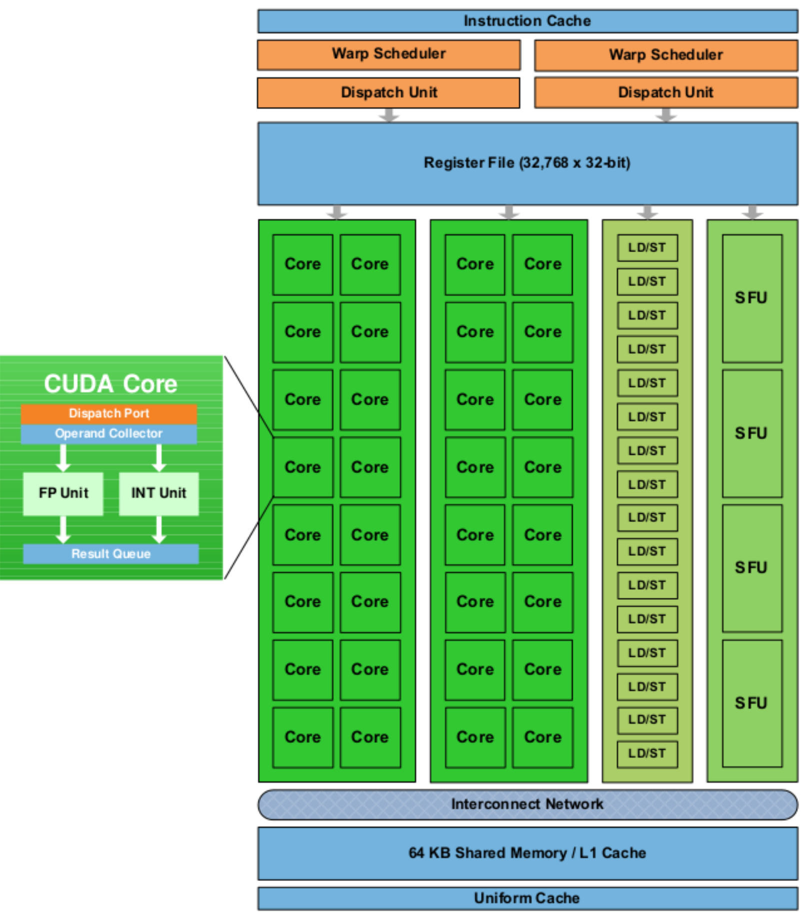

Figura 2.11: Diagrama de bloques de un Streaming Multiprocessor. (Recurso obtenido de [NVIDIA15])

El desarrollo de aplicaciones para la GPU se realiza mediante una plataforma de cómputo paralelo llamada CUDA. CUDA fue desarrollada por la compañía estadounidense llamada nVidia, en 2006. Esta plataforma permite compilar código en lenguaje C, C++ y Fortran, para su ejecución en las GPU, permitiendo así el desarrollo de aplicaciones de propósito general en las GPU.

A pesar del alto poder de cómputo con el que cuentas las GPU estas suelen ser usadas en conjunto con los CPU para el desarrollo de aplicaciones debido a que todos los procesos del sistema operativo de una computadora se ejecutan en el CPU. Para aplicaciones de alta demanda de procesamiento la GPU puede ser usada como un coprocesador, encargándose de procesar grandes volúmenes de datos y aminorando la carga de trabajo en el CPU.

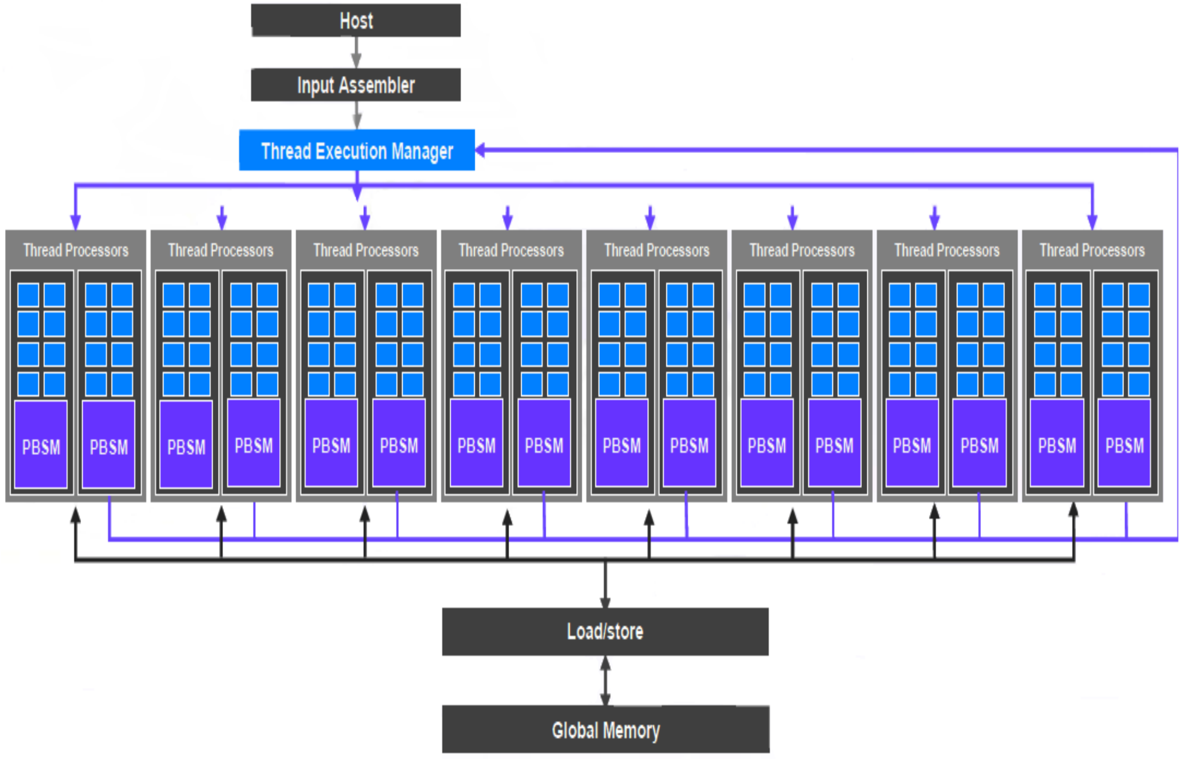

Figura 2.12: Diagrama de alto nivel de una GPU. (Recurso obtenido de [NVIDIA15])

Actualmente las GPU (en conjunto con los CPU) son utilizadas para el desarrollo de aplicaciones en diversas áreas de las ciencias, como son: aprendizaje máquina, procesamiento digital de imágenes, álgebra lineal, estadística, reconstrucción 3D, y en general, aplicaciones que demanden el procesamiento de grandes volúmenes de datos en tiempos relativamente cortos.

2.1.4. Sistemas Embebidos y Cómputo Heterogéneo

El término *Sistema Embebido* se refiere a un sistema de cómputo dedicado a realizar una o varias funciones específicas, frecuentemente en un sistema de tiempo real, [Hea03]. Los sistemas embebidos comúnmente forman parte de sistemas mucho mas grandes que, además de contar con otros componentes electrónicos, pueden contar con partes mecánicas.

A diferencia de las computadoras de propósito general, los sistemas embebidos poseen características que los hacen superiores en aplicaciones específicas:

- Menor consumo de energía.
- Menor tamaño.
- Rangos de operación robustos.
- Menor costo.

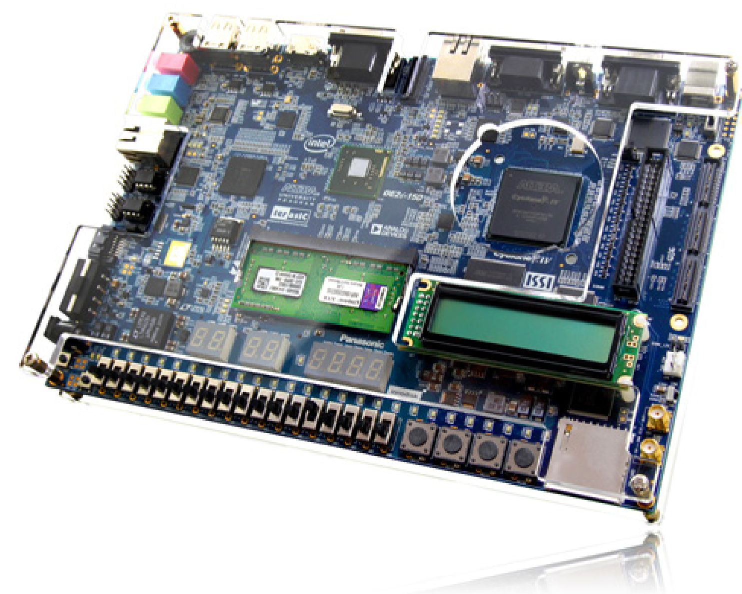

Figura 2.13: Tarjeta de desarrollo Intel DE2i-150. Cuenta con un procesador Intel Atom y un FPGA Altera Cyclone IV. (Recurso obtenido de [Terasic13])

Actualmente, los sistemas embebidos se pueden encontrar en diversos dispositivos de uso cotidiano, que van desde dispositivos portátiles (reproductores mp3, relojes digitales, etc.) hasta sistemas muy complejos (vehículos híbridos, robots, controles de fábrica, entre otros) que además cuentan con interfaces para el envío y recepción de datos (USB, Ethernet, RS232, etc).

Así mismo, existen plataformas de hardware que permiten desarrollar prototipos de sistemas embebidos, estas plataformas pueden ser programadas mediante lenguaje ensamblador o mediante lenguaje C, cuentan con periféricos para envío y transmisión de datos, memorias para almacenar información y un microprocesador. Ejemplos de estos sistemas son las plataformas de hardware Arduino, RaspBerry, Intel Galileo, entre otras. También existen plataformas con un FPGA integrado que permiten describir hardware para prototipos. Estas plataformas suelen contar con un procesador de propósito general permitiendo así el desarrollo de aplicaciones en el procesador y el desarrollo de un sistema embebido o coprocesador en el FPGA. Un ejemplo de estas tarjetas es la DE2i-150 de Intel, la cuál cuenta con un procesador Intel Atom y un FPGA Altera Cyclone IV comunicados a través de un puerto PCI Express (Figura 2.13).

En plataformas de desarrollo, tales como la tarjeta DE2i-150, y en sistemas que combinan distintas arquitecturas de hardware (tales como aquellos en donde se utiliza tanto un CPU como un GPU) los diseñadores de aplicaciones se enfrentan a un gran reto: diseñar aplicaciones que obtengan el mayor beneficio de cada arquitectura para lograr un objetivo común. Esto implica solucionar problemas como: comunicación entre arquitecturas, sincronización, formato de datos, entre otros. A este tipo de sistemas, que involucran más de un tipo de arquitectura, se le conoce como sistemas de Cómputo Heterogéneo. Los sistemas de Cómputo Heterogéneo son sistemas multiprocesador que mejoran el rendimiento no por tener varios procesadores, sino por el hecho de designar tareas específicas a cada uno de ellos (tareas en donde se obtenga el mejor desempeño de un procesador).

En [BDHHS10] se realiza el estudio de tres arquitecturas de hardware (FPGA, GPU y CPU) y se definen el tipo de aplicaciones que aprovechan el mejor desempeño de cada una, en tal trabajo se concluye lo siguiente:

- Los problemas orientados a grandes cantidades de datos de coma flotante deben ser atacados haciendo uso de las GPU.
- Para tareas de coordinación entre arquitecturas, paso de mensajes y toma de decisiones es mejor usar una CPU.
- Para la aceleración de problemas no paralelizables o procesamiento de señales los FPGA son la opción a elegir.

En [Kjaer10] se desarrollan una serie de sistemas de cómputo heterogéneo como robots y vehículos autónomos combinando tres arquitecturas: CPU Multicore, FPGA y GPU. Para la adquisición de datos e implementación de algoritmos de pre-procesamiento hace uso de los FPGA, para el cómputo masivo utiliza una GPU y finalmente, para la toma de decisiones, mediante algoritmos de inteligencia artificial, hace uso de un CPU Multicore.

2.2. Cámaras RGB-D

Las cámaras RGB-D basan su funcionamiento en la proyección de un patrón lineal sobre la escena a observar, la misma cámara detecta las deformaciones que este patrón tenga en la escena y en base a esto obtienen la información 3D de los objetos. Estas cámaras están compuestas por un proyector de luz estructurada, un detector especial para la luz proyectada y una cámara digital RGB.

La cámara Primesensor de Primesense proyecta un patrón infrarrojo conocido en la escena (Figura 2.14), que permite determinar medidas de profundidad de cada pixel en una imagen a color con el análisis de la deformación de dicho patrón (cámara que detecta luz infrarroja). La resolución de esta cámara es de 640x480 pixeles a 30 cuadros por segundo, con un campo de visión de horizontal de 57º, vertical de 43º y un rango de distancia de 40 centímetros a 6 metros.

Figura 2.14: Patrón infrarrojo proyectado por una cámara RGB-D. Se proyecta el patrón en forma de puntos y la cámara observa la deformación de los puntos en la escena para generar información 3D.

Las cámaras RGB-D brindan información de profundidad que puede ser analizada para determinar no solo la distancia a la que se encuentra los elementos de la imagen, sino también la posición en coordenadas tridimensionales con referencia a la posición de la cámara.

La adquisición de datos desde una cámara RGB-D puede ser realizada mediante un software llamado OpenNI. OpenNI permite realizar la extracción de esta información de manera automática en cámaras basadas en la tecnología de proyección de luz estructurada. La información adquirida se presenta en forma de *Nubes de Puntos* organizadas de acuerdo a los valores de sus coordenadas *(x,y,z)* en el mundo real, y de acuerdo a los valores de coordenadas *(fila,columna)* de una imagen digital (Figura 2.15).

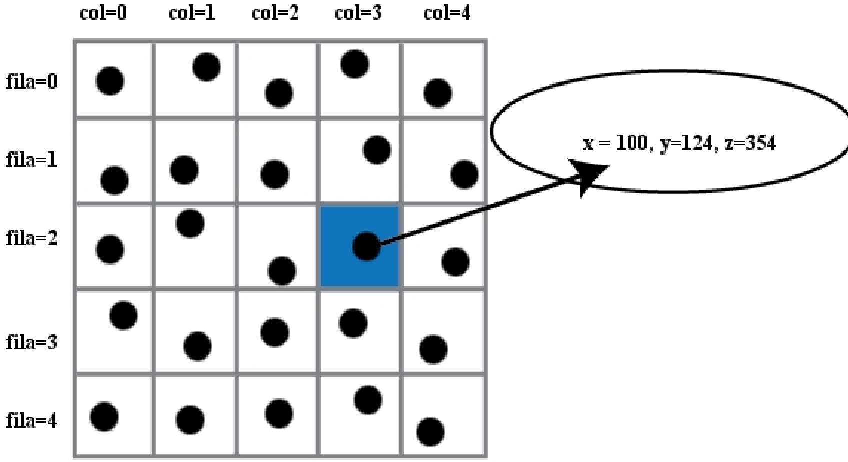

Figura 2.15: Representación de la organización de una nube de puntos. Se organizan de acuerdo a los valores de fila,columna y de acuerdo a los valores de sus coordenada x,y,z.

Capítulo 3. Descripción del problema

Uno de los objetivos principales en el área de robótica de servicio es lograr que un robot realice tareas de asistencia doméstica de manera autónoma, segura y con tiempos de respuesta inmediatos. Un problema comúnmente abordado en esta área consiste en la detección de objetos (alimentos, herramientas, recipientes, etc.) sobre superficies planas (mesas, aparadores, lacenas, etc.) para su posterior reconocimiento y manipulación. El cumplimiento de esta actividad (detección de objetos sobre superficies planas) le permite a los robots la ejecución de otras múltiples actividades con mayor grado de complejidad.

En el presente capítulo se describe el problema planteado en esta tesis: detección de objetos sobre superficies planas mediante técnicas de procesamiento de nubes de puntos. Así mismo, se describen los algoritmos implementados para dar solución a dicha tarea.

3.1. Planteamiento

Como se mencionó anteriormente, el principal objetivo de la robótica de servicio es fabricar robots auxiliares que ayuden a las personas en tareas cotidianas del hogar (limpieza, atención, cuidados, etc.) y una de las principales tareas en esta área consiste en la manipulación de objetos. Un robot capaz de manipular objetos puede realizar múltiples funciones como: hacer entrega de medicamentos, alimentos, abrir y cerrar puertas, accionar interruptores, etc. Sin embargo, para lograr la manipulación de objetos existe un paso previo que debe llevar a cabo el robot: la detección de objetos, en este trabajo se aborda este problema; en específico, la detección de objetos sobre superficies planas como mesas.

El problema de detección de objetos sobre una superficie plana, en una nube de puntos N, consiste en obtener un total de M conjuntos: $\mathbf{S_1}$, $\mathbf{S_2}$, $\mathbf{S_3}$, ..., $\mathbf{S_M}$, con las siguientes propiedades:

$$(\mathbf{S_1} \cup \mathbf{S_2} \cup \mathbf{S_3} \cup \ldots \cup \mathbf{S_M}) \subseteq N$$

$$(\mathbf{S_1} \cap \mathbf{S_2} \cap \mathbf{S_3} \cap \ldots \cap \mathbf{S_M}) = \emptyset$$

Donde cada conjunto S es una nube de puntos perteneciente a un objeto de la escena N y donde la diferencia de conjuntos:

$$N - (\mathbf{S_1} \cup \mathbf{S_2} \cup \mathbf{S_3} \cup \ldots \cup \mathbf{S_M})$$

son puntos pertenecientes a la superficie plana sobre la cual se encuentran ubicados los objetos y puntos que se encuentran debajo de dicha superficie plana. Lo anterior se ilustra gráficamente en la Figura 3.1.

Para dar solución al problema planteado en este trabajo, es necesario llevar a cabo dos distintas etapas de procesamiento:

- Extracción de puntos característicos. Consiste en obtener características locales para cada elemento perteneciente al conjunto ***N***, con la finalidad de obtener información geométrica de cada elemento.
- Segmentación de la superficie plana dominante. Consiste en calcular un modelo matemático ***K*** el cual describa a la mayor cantidad de elementos de ***N*** pertenecientes a la misma superficie plana en la escena, posteriormente se procede a suprimir de la escena todos los puntos que empaten con el modelo ***K***.

A continuación se detallan los algoritmos implementados en cada una de las etapas de procesamiento descritas en el párrafo anterior.

Figura 3.1: Robot móvil de servicio Justina intentando detectar objetos situados sobre una superficie plana.

3.2. Extracción de puntos característicos

En visión por computadora y procesamiento de imágenes, un punto característico se define como un extracto de información relevante correspondiente a un punto en una escena. La información que contiene un punto característico puede ser: información de color, textura, forma, orientación, entre otras. En procesamiento 3D, se hace uso de **vectores normales** como puntos característicos de una escena 3D debido a que representan importantes propiedades geométricas de una superficie y sirven para extraer información semántica de los datos 3D obtenidos por un sensor. En este trabajo, la información obtenida mediante los **vectores normales** será usada para agrupar aquellos puntos con propiedades geométricas similares (puntos pertenecientes a la misma superficie).

Un método comúnmente usado para el calculo del vector normal de un punto $\boldsymbol{p_i}$ consiste en:

- Obtener una vecindad de puntos $\boldsymbol{P_i}$ respecto al punto $\boldsymbol{p_i}$. Esta vecindad $\boldsymbol{P_i}$ puede estar dada por los $\boldsymbol{k}$ vecinos más cercanos a $\boldsymbol{p_i}$ (con base en su distancia euclidiana) o por los puntos localizados dentro de un radio $\boldsymbol{r}$ de distancia desde $\boldsymbol{p_i}$.
- Posteriormente se obtiene el vector normal característico a la vecindad mediante un método como PCA o LSM.

El anterior, a pesar de ser un método muy efectivo demanda mucho poder de cómputo (como demuestran [HHRB11] en sus estudios). Una alternativa consiste en encontrar vecindades de puntos respecto a los valores de *fila,columna,* no respecto a los valores *x,y,z*, para posteriormente calcular el vector normal relativo a la vecindad hallada.

El método para calcular vectores normales sobre una nube de puntos, implementado en este trabajo, consiste en:

- Hallar la vecindad de puntos usando las coordenadas de una imagen digital (fila, columna). Con esto se evita realizar una búsqueda en todo el espacio 3D.
- Calcular el vector normal utilizando un método de bajo costo computacional.

Como se mencionó en el apartado 2.2, los sensores RGB-D organizan las nubes de puntos de dos formas: con base en el valor de sus coordenadas *x,y,z* y con base en sus valores de *fila,columna.* Lo anterior permite encontrar vecindades de puntos de manera más rápida debido a que la búsqueda se reduce a una búsqueda en dos dimensiones con rangos fijos (en lugar de una búsqueda en tres dimensiones).

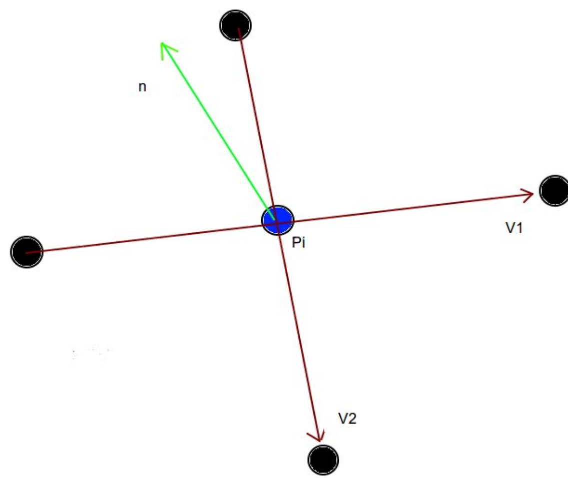

Figura 3.2: Vecindad Pi (puntos negros) relativa a un punto pi (punto azul), se obtienen dos vectores V1 y V2, y se calcula el producto cruz entre ellos para obtener el vector normal n.

Una vez hallada la vecindad de un punto $\boldsymbol{p_i}$, esta puede ser vista como una superficie. Un método de bajo costo computacional para hallar el vector normal ***n*** (correspondiente a la superficie formada por la vecindad del punto $\boldsymbol{p_i}$) consiste en hallar dos vectores (***V1*** y ***V2***) tangenciales a dicha superficie y realizar un *producto cruz*. Esta idea se encuentra ilustrada en la Figura 3.2.

El producto cruz de dos vectores **a** y **b** se representa por **a** x **b** y se define por la expresión:

$$a\,x\,b = (|a||b|\sin(\theta))\hat{n}$$

donde, de acuerdo con [KH06], **n** es un vector ortogonal a **a** y **b** y a la superficie determinada por estos, y **θ** corresponde al ángulo de separación entre **a** y **b**. En otras palabras, el vector resultante de un producto cruz es un vector perpendicular (**vector normal**) a los dos vectores involucrados en la operación.

El Algoritmo 3.1 muestra los pasos necesarios para el cálculo de vectores normales en la nube de puntos. En el algoritmo se realiza el cálculo de vecindad con base en los valores *fila,columna* de cada punto en la nube de puntos.

3.3. Segmentación de la superficie plana dominante

En robótica de servicio es importante que un robot perciba correctamente la geometría del entorno que lo rodea. Esta característica es un requisito crucial para que el robot se desempeñe adecuadamente en entornos dinámicos. Para el problema particular planteado en este trabajo se requiere completar tres pasos:

- Detectar la superficie plana de mayor tamaño en una escena 3D.
- Obtener el modelo matemático de la superficie plana hallada.
- Suprimir, de la escena, los puntos pertenecientes a la superficie plana de mayor tamaño y todos los puntos objetos ubicados por debajo de dicha superficie.

De esta forma en la escena únicamente permanecerán aquellos puntos pertenecientes a los objetos ubicados por encima de la superficie plana dominante. En la Figura 3.3 se puede observar, de manera gráfica, el proceso anteriormente descrito.

Algoritmo 3.1: Cálculo de vectores normales en una nube de puntos organizada.

1. Sea **P** una nube de puntos organizada en **n** filas y **m** columnas.
2. Sea **N** un conjunto de vectores normales organizados en **n** filas y **m** columnas.
3. Para todo **i** = 0, 1, 2, … **n**
4. Para todo **j** = 0, 1, 2, … **m**
5. Obtener los vectores tangenciales a la superficie de la vecindad respecto al punto **P**(**i**,**j**).
   vector A = **P**(**i**+1,**j**) – **P**(**i**-1,**j**)
   vector B = **P**(**i**,**j**+1) – **P**(**i**,**j**-1)
6. Realizar el producto cruz entre los vectores tangenciales para obtener el vector normal correspondiente al punto **P**(**i**,**j**).
   N(i , j) = **A** x **B**

Existen diversos estudios sobre segmentación de superficies planas dominantes:

- En [HHRB11] calculan los vectores normales de cada punto en la escena y posteriormente agrupan (mediante algoritmos de *clustering*) aquellos puntos cercanos entre sí y cuyos vectores normales contienen características similares.
- En [WLQX12] se divide la escena en vecindades y para cada vecindad obtienen un modelo matemático (el cuál describe la superficie plana dominante de dicha vecindad). Posteriormente se agrupan aquellas vecindades cuyos modelos sean similares obteniendo así todas las superficies planas de el escena.

- En [Hyun13] se calculan los vectores normales de cada punto y se agrupan en distintos conjuntos de acuerdo a sus propiedades algebraicas (aquellos puntos con la misma inclinación forman parte del mismo conjunto). El conjunto con mayor número de elementos representan una superficie plana.
- En [B2011] se utiliza el algoritmo de Transformada de Hough en 3D para hallar planos en una escena 3D.
- En [FB81] se hace uso del algoritmo de Random Sample Concensus (RANSAC) para esta misma tarea.

En este trabajo se hace uso del algoritmo RANSAC para llevar a cabo la detección del plano dominante de una escena 3D. En el Algoritmo 3.2 se detalla los pasos a seguir para aplicar RANSAC sobre una escena 3D.

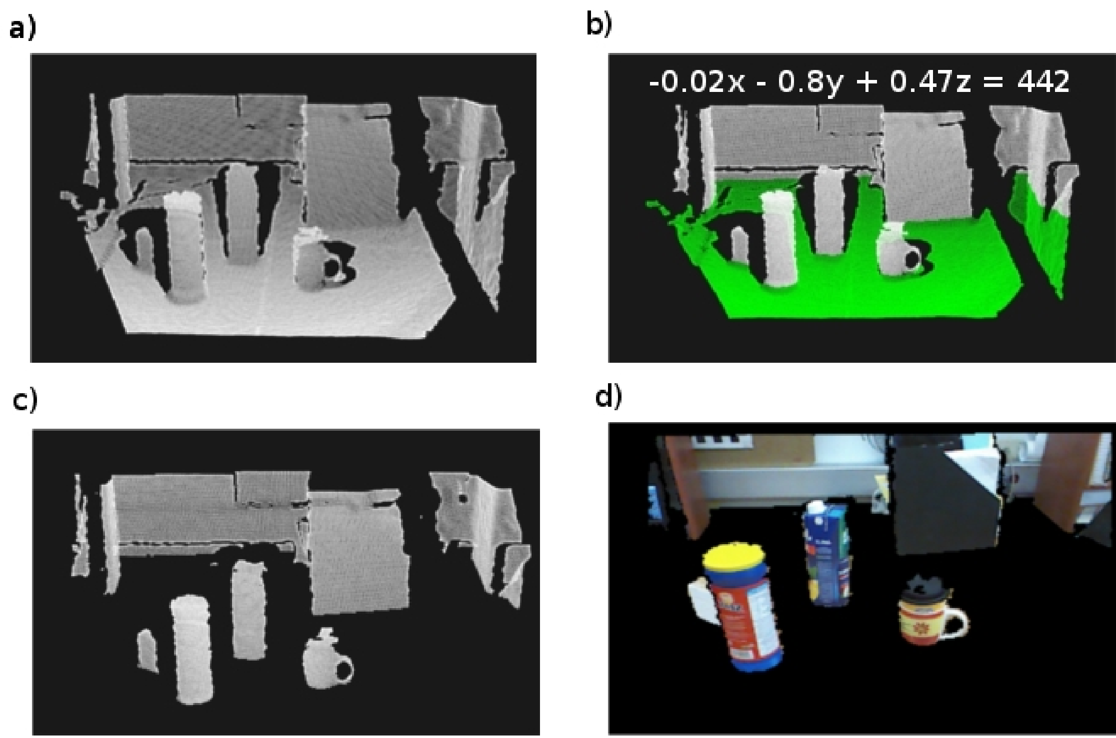

Figura 3.3: Proceso de segmentación de la superficie plana dominante. a) Escena 3D en la cual se puede observar un conjunto de puntos que pertenecen a diversos objetos ubicados sobre una superficie plana. b) Modelo matemático que representa al conjunto de puntos que conforman la superficie plana dominante (verde). c) La superficie plana dominante ha sido removida de la escena, únicamente prevalecen aquellos puntos pertenecientes a objetos ubicados por encima de la superficie plana dominante. d) Escena 2D de la cual se han suprimido todos aquellos puntos pertenecientes a la superficie plana dominante.

Algoritmo 3.2: RANSAC aplicado a una escena 3D.

1. Parámetros iniciales:
   P: Nube de puntos.
   I: Conjunto de *inliers*.
   M: Modelo matemático.
   S: Conjunto de muestreo mínimo.
   d: Umbral máximo de desviación estándar.
   bestI: Mejor conjunto de *inliers*.
   bestD: Mejor desviación estándar.

2. Hacer:
   bestD = +∞
   bestI = Ø

3. Elegir de manera aleatoria tres elementos del conjunto **P** y almacenarlos en **S**.
   (Muestreo mínimo requerido para calcular la ecuación de una superficie plana)

4. Calcular **M** con base en **S**.

5. Calcular **I** con base en el nuevo valor de **M**.
   (Obtener aquellos elementos de **P** cercanos a la superficie plana descrita por **M)**

6. Si **I** > **bestI** OR (**I**==**bestI** AND desviación_estándar (**M**,**P**)<**bestD**)
   bestD = desviación_estándar (**M**,**P**)
   bestI = **I**

7. Repetir desde el paso 3 hasta que **bestD** <= **d**

Capítulo 4. Esquema de procesamiento en CPU

Para llevar a cabo la ejecución de un algoritmo en un CPU son necesarias una serie de etapas que van desde la traducción del algoritmo (de pseudocódigo a código en un lenguaje de programación) hasta la ejecución del programa generado (programa de computadora).

En primer lugar se debe elegir un método de ejecución. Un programa de computadora puede ser ejecutado por el CPU de dos maneras distintas:

- Ejecutar el programa directamente en el CPU (ejecución directa).
- Utilizar un software intermedio entre el programa a ejecutar y el CPU (ejecución mediante sistema operativo).

La Tabla 4.1 muestra una comparativa entre ambos tipos de ejecución.

Tipo de ejecución	Tiempo de ejecución	Consumo de recursos del CPU	Tiempo de desarrollo del programa	Mantenimiento del programa	Portabilidad del programa
Ejecución directa	Relativamente corto	Varia con base en el manejo de los recursos por parte del programador	Largo (requiere el uso de un lenguaje de bajo nivel o lenguaje máquina)	Complicado	No portable
Ejecución mediante sistema operativo	Varia conforme a la carga de trabajo del CPU	Varia conforme al número de procesos en ejecución	Relativamente corto (requiere el uso de un lenguaje de alto nivel)	Sencillo	Portable

Tabla 4.1: Comparativa entre tipos de ejecución.

Una vez elegido el método de ejecución se procede a elegir el lenguaje de programación al cual se traducirá el algoritmo escrito en pseudocódigo. Esta elección debe realizarse tomando en cuenta diversos factores: paradigma de programación, velocidad del programa generado, curva de aprendizaje, portabilidad de los programas, entre otras.

En este trabajo, para el desarrollo y ejecución de los algoritmos en el CPU, se hace uso del método de ejecución mediante sistema operativo; lo anterior debido al relativamente corto tiempo de desarrollo requerido y a la necesidad de portabilidad en los programas desarrollados. Así mismo el lenguaje de programación utilizado para llevar a cabo la traducción de los algoritmos planteados es C++, debido a la portabilidad de los programas desarrollados en este lenguaje. Finalmente, el sistema operativo elegido para llevar a cabo la ejecución es Ubuntu 14.04.

En los apartados siguientes se presentan diversos diagramas (representados mediante Lenguaje Unificado de Modelado, UML por sus siglas en inglés) dichos diagramas muestran la estructura lógica de los programas desarrollados en C++.

4.1. Diagramas de clases

Los diagramas de clases representan la estructura interna de un programa bajo el paradigma orientado a objetos. En estos diagramas se describe como están conformadas cada una de las clases y las relaciones existentes entre ellas. En el presente apartado se muestran los diagramas de clases pertenecientes a la implementación en C++ de los procesos descritos en la sección 3.1.

4.1.1. Extracción de puntos característicos.

En la Figura 4.1 se muestra el diagrama de clases correspondiente a la implementación del algoritmo de extracción de puntos característicos de una nube de puntos. Este proceso se encarga de realizar el cálculo de los vectores normales para cada punto en una escena 3D. El programa está conformado por un total de 6 clases:

- *coordenada_3D*: clase abstracta que representa tres coordenadas (x,y,z) en un sistema de coordenadas de tres dimensiones.
- *punto_3D*: clase que representa un punto en un sistema de coordenadas de tres dimensiones.
- *vector_3D*: clase que representa un vector en un sistema de coordenadas de tres dimensiones.

- *nube_puntos_XYZ*: clase que representa un conjunto de puntos (de la clase punto_3D) en un sistema de coordenadas de tres dimensiones.
- *calculos_comunes*: clase que contiene diversas operaciones matemáticas aplicables a vectores y puntos 3D.
- *calculo_normales*: clase mediante la cual se inicia el proceso de calculo de los vectores normales sobre una nube de puntos.

4.1.2. Segmentación de la superficie plana dominante.

El diagrama de clases correspondiente a la implementación del algoritmo RANSAC se muestra en la Figura 4.2. Dicho diagrama esta conformado por las siguientes clases y métodos:

- *ransac_3d*: clase mediante la cual se inicia el proceso de segmentación de la superficie plana dominante mediante el algoritmo RANSAC. Esta clase cuenta con 4 propiedades y 5 métodos para llevar a cabo la ejecución de dicho algoritmo:
- *conjunto_muestreo_mínimo* (propiedad): almacena el conjunto de muestreo mínimo necesario (3 puntos 3D) para obtener el modelo matemático que describe una superficie plana.
- *ecuacion_plano* (propiedad): almacena el modelo matemático correspondiente a una superficie plana.
- *conjunto_inliers* (propiedad): almacena el conjunto de puntos 3D que son descritos por el modelo almacenado en *ecuacion_plano*.
- *nube_puntos_segmentada* (propiedad): almacena la nube de puntos obtenida al suprimir los puntos correspondientes a la superficie plana dominante de la nube de puntos original.
- *muestreo_minimo_aleatorio* (método): obtiene aleatoriamente 3 puntos almacenados en *conjunto_inliers*.
- *obtener_ecuacion_plano* (método): calcula la ecuación de la superficie plana dominante mediante *conjunto_muestreo_minimo* y la almacena en *ecuacion_plano*.
- *obtener_inliers* (método): obtiene un conjunto de puntos pertenecientes a la superficie plana dominante descrita por *ecuacion_plano* y lo almacena el *conjunto_inliers*.
- *obtener_nube_puntos_segmentada* (método): suprime los puntos pertenecientes a la superficie plana dominante de la escena original.
- Las descripciones de las clases *coordenada_3D*, *punto_3D*, *vector_3D*, *nube_puntos_XYZ* y *calculos_comunes* corresponden con las del apartado anterior (extracción de puntos característicos).

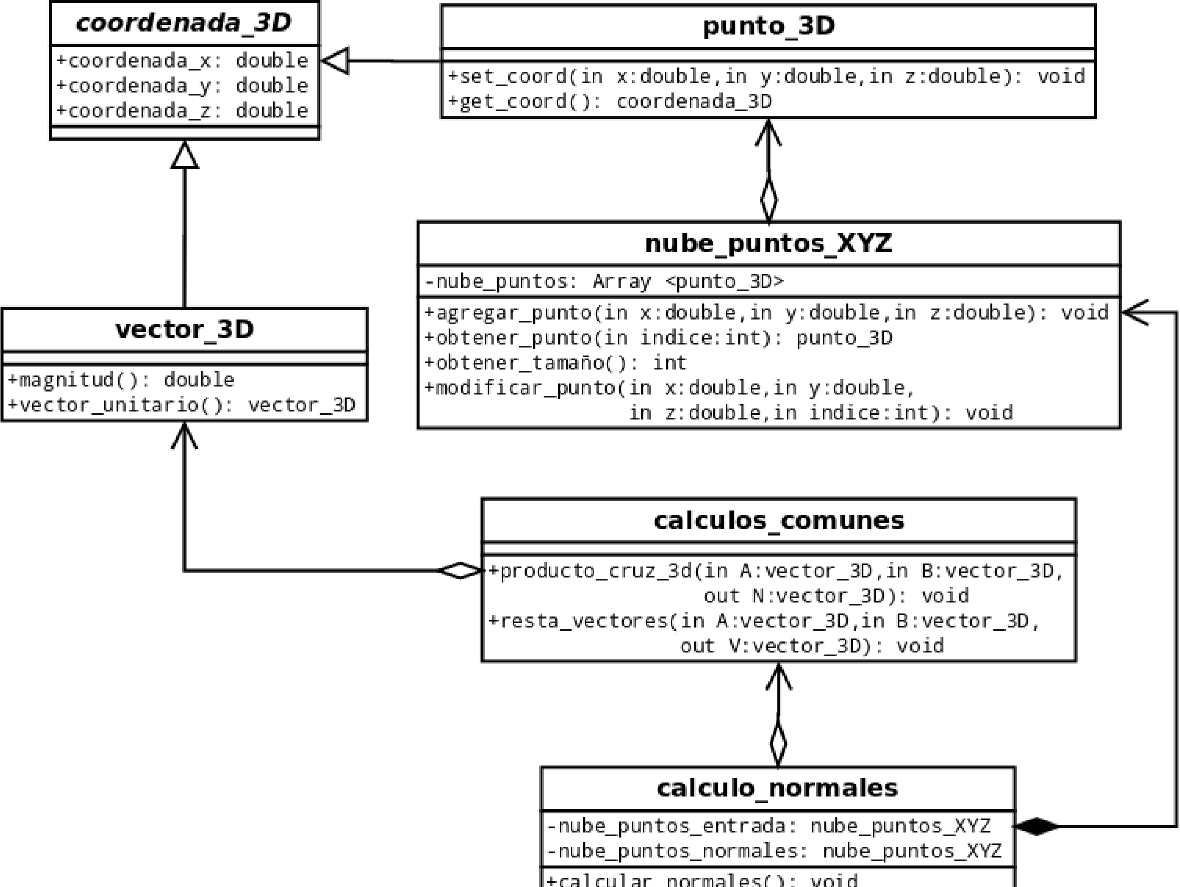

Figura 4.1: Diagrama de clases (extracción de puntos característicos).

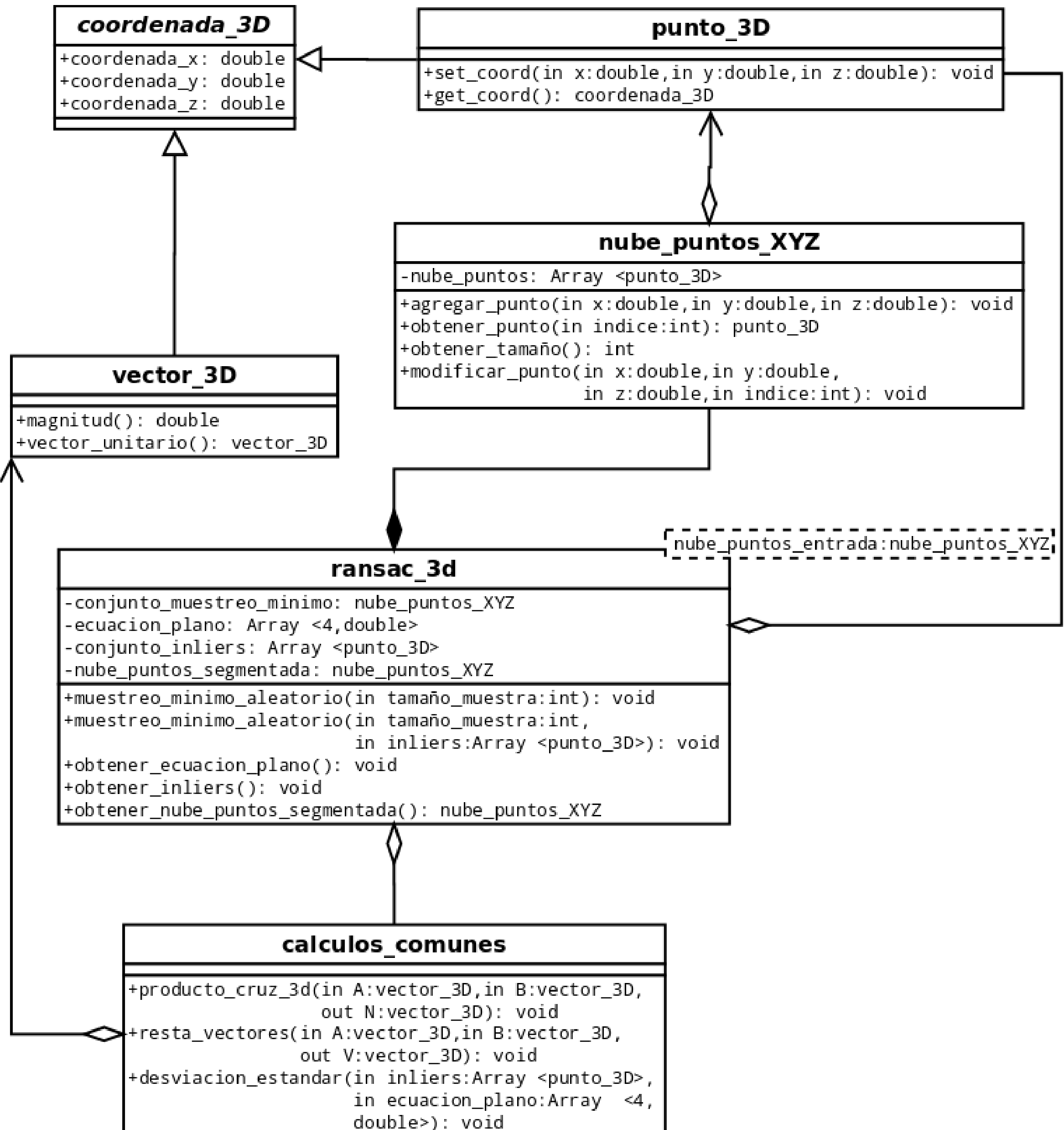

Figura 4.2: Diagrama de clases (segmentación de la superficie plana dominante).

4.2. Diagramas de flujo

Los diagramas de flujo, como son descritos en [Harris00], son diagramas que muestran de forma visual la información de un proceso. Información como: pasos de ejecución, funciones, etc., son mostradas de manera organizada (cronológica o secuencial). En UML los diagramas de flujo son conocidos como diagramas de actividades y son utilizados para modelar procesos computacionales y organizacionales.

Los diagramas de actividades están compuestos por distintos elementos gráficos que ayudan a representar el flujo de control de un proceso en su totalidad (en la Tabla 4.2 se describen estos elementos a detalle). A diferencia de los diagramas de clase, los cuales representan la estructura de un proceso, los diagramas de actividades describen el comportamiento de dicho proceso. En el presente apartado se muestran los diagramas de actividades correspondientes a los algoritmos 3.1 y 3.2, estos diagramas hacen uso de la estructura descrita por los diagramas de clase presentados en el apartado 4.1.

4.2.1. Extracción de puntos característicos

La Figura 4.3 muestra el diagrama de flujo correspondiente al proceso de extracción de puntos característicos sobre una escena 3D. Las variables y métodos utilizados fueron extraídos del diagrama de clase de la Figura 4.1, a continuación se detalla su funcionamiento:

- **nube_puntos_entrada** (abreviada como variable **P**): Variable que almacena la nube de puntos correspondiente a la escena 3D.
- **nube_puntos_normales** (abreviada como variable **N**): Variable que almacena el conjunto de vectores normales que serán calculados a partir del conjunto **P**.
- **A, B**: Variables que almacenan los vectores ortogonales al vector normal por calcular.
- **i, j**: Variables utilizadas para indexar a cada elemento del conjunto **P** por medio de sus valores fila-columna.
- **n**, **m**: Constantes que indican los valores máximos de fila y columna, respectivamente, para acceder a los datos de la variable **P**.
- **resta_vectores**: Subrutina que calcula la resta de dos vectores. Esta subrutina recibe como primeros dos parámetros los vectores a restar y, como tercer parámetro, la variable en la cual se almacenará el resultado.
- **producto_cruz_3D**: Subrutina que calcula el producto cruz entre dos vectores. Recibe como primeros dos parámetros los vectores a operar, y como tercer parámetro, la variable en la cual se almacenará el resultado.

Asimismo se detalla a continuación el flujo proceso representado en el diagrama de la Figura 4.3:

1. Se realiza la adquisición de la nube de puntos y se almacena en la variable **P**.
2. Se asigna un conjunto vacío a la variable **N**. Más adelante esta variable almacenará los vectores normales calculados.
3. Se inicializan las variables **i** y **j** con un valor de 0. Posteriormente se modificará el valor de estas variables para acceder a cada elemento de las variables **P** y **N**.

Elemento	**Descripción**
Inicio/Fin	Indican el inicio o fin de un proceso.
Entrada/Salida	Representa cualquier tipo de introducción de datos en la memoria desde los periféricos de entrada. Asimismo representa la carga de datos en los periféricos de salida.
Proceso	Describe la realización de operaciones aritméticas y de operaciones de asignación de datos.
Proceso predefinido	Describe la ejecución de un módulo independiente del programa principal. Recibe una entrada desde el programa principal, realiza una tarea determinada y puede devolver un valor de salida.
Decisión	Plantea la selección de una alternativa con base en la evaluación de una condición.
Línea de flujo	Indican el sentido de ejecución de las operaciones.

Tabla 4.2: Simbología correspondiente a un diagrama de actividades.

4. Se realiza un recorrido fila-columna sobre la variable **P**. Esto se logra incrementando de uno en uno los valores de **i** y **j**, y condicionando que dichos valores no sobrepasen los límites **n** y **m**.
5. Para cada elemento accedido (mediante fila-columna) de la variable **P**. Se calculan los vectores **A** y **B** y se calcula un vector normal (véase el apartado 3.2).
6. El vector normal calculado se almacena posteriormente en la variable **N**.

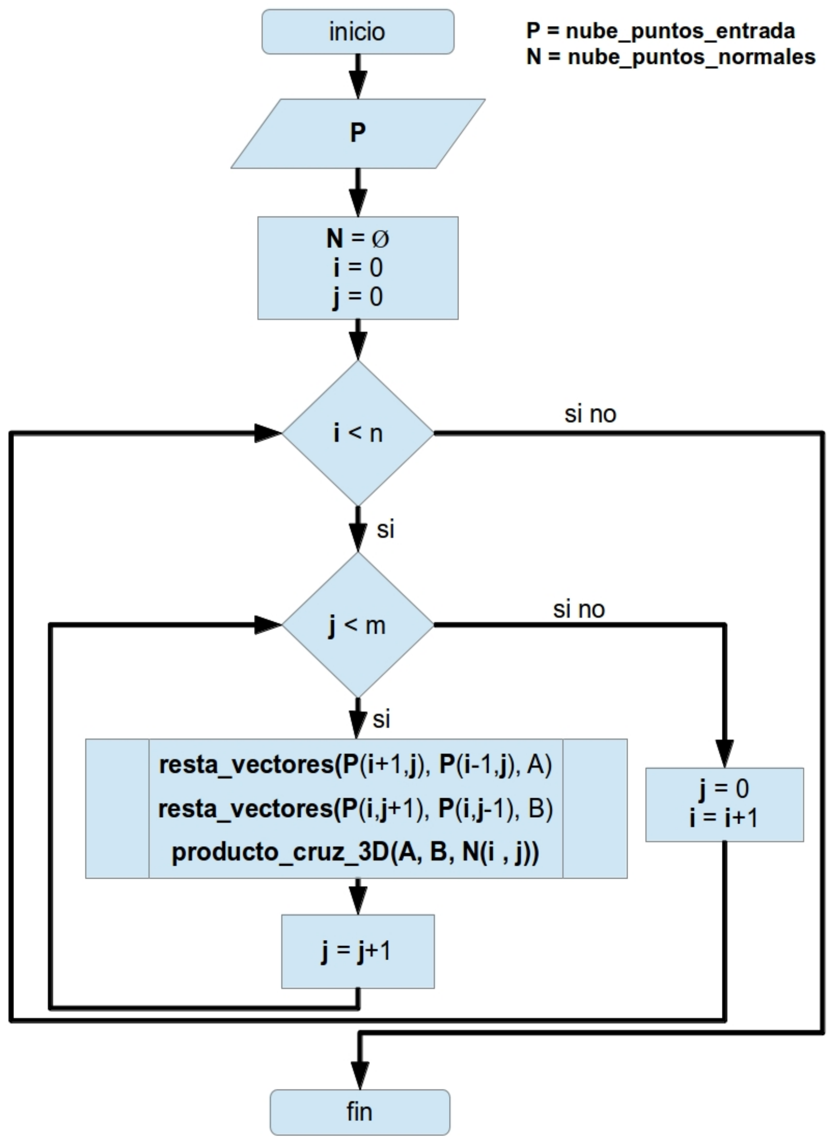

Figura 4.3: Diagrama de flujo correspondiente al proceso de extracción de puntos característicos sobre una escena 3D.

Al finalizar este proceso, la variable **P** (**nube_puntos_entrada**) continúa almacenando la nube de puntos original. Por otro lado, la variable **N** (**nube_puntos_normales**) almacena el conjunto de todos los vectores normales calculados a partir de **P**.

4.2.2. Segmentación de la superficie plana dominante

En la Figura 4.4 se muestra el diagrama de flujo correspondiente al algoritmo RANSAC aplicado a una escena 3D. Las variables y métodos utilizados fueron extraídos del diagrama de clase de la Figura 4.2, a continuación se detalla la funcionalidad de cada uno:

- **nube_puntos_entrada** (abreviada como variable **P**): Variable que almacena la nube de puntos correspondiente a la escena 3D.
- **conjunto_muestreo_minimo** (abreviada como variable **S**): Variable que almacena el conjunto de datos mínimo necesario para obtener la ecuación de una superficie plana (3 datos).
- **ecuacion_plano** (abreviada como variable **M**): Variable que almacena la ecuación de la superficie plana obtenida en cada iteración del algoritmo.
- **conjunto_inliers** (abreviada como variable **I**): Variable que almacena el conjunto de puntos, pertenecientes a **P**, que empatan con la ecuación almacenada en **M**.
- **D**: Umbral para determinar si la ecuación del plano hallada corresponde al plano dominante de la escena.
- **bestI**: En esta variable se almacena el mejor conjunto de inliers obtenidos durante la ejecución del algoritmo.
- **bestD**: Esta variable almacena el valor de la mejor desviación estándar obtenida durante la ejecución del algoritmo.
- **muestreo_minimo_aleatorio**: Subrutina que obtiene, de manera aleatoria, un subconjunto de puntos de **P**; recibe como parámetro el tamaño del conjunto deseado. El conjunto obtenido se almacena en la variable **S**.
- **obtener_ecuacion_plano**: Subrutina que calcula la ecuación de un plano a partir del conjunto **S**. La ecuación calculada se almacena en la variable **M**.
- **obtener_inliers**: Subrutina que encuentra todos aquellos puntos, pertenecientes a **P**, que empatan con la ecuación del plano almacenada en **M**; los puntos hallados (inliers) se almacenan en la variable **I**. El diagrama de flujo correspondiente a esta subrutina se muestra en la Figura 4.5.
- **desviacion_estandar**: Subrutina que calcula la desviación estándar entre la superficie plana hallada (**M**) y los puntos de la escena (**P**).

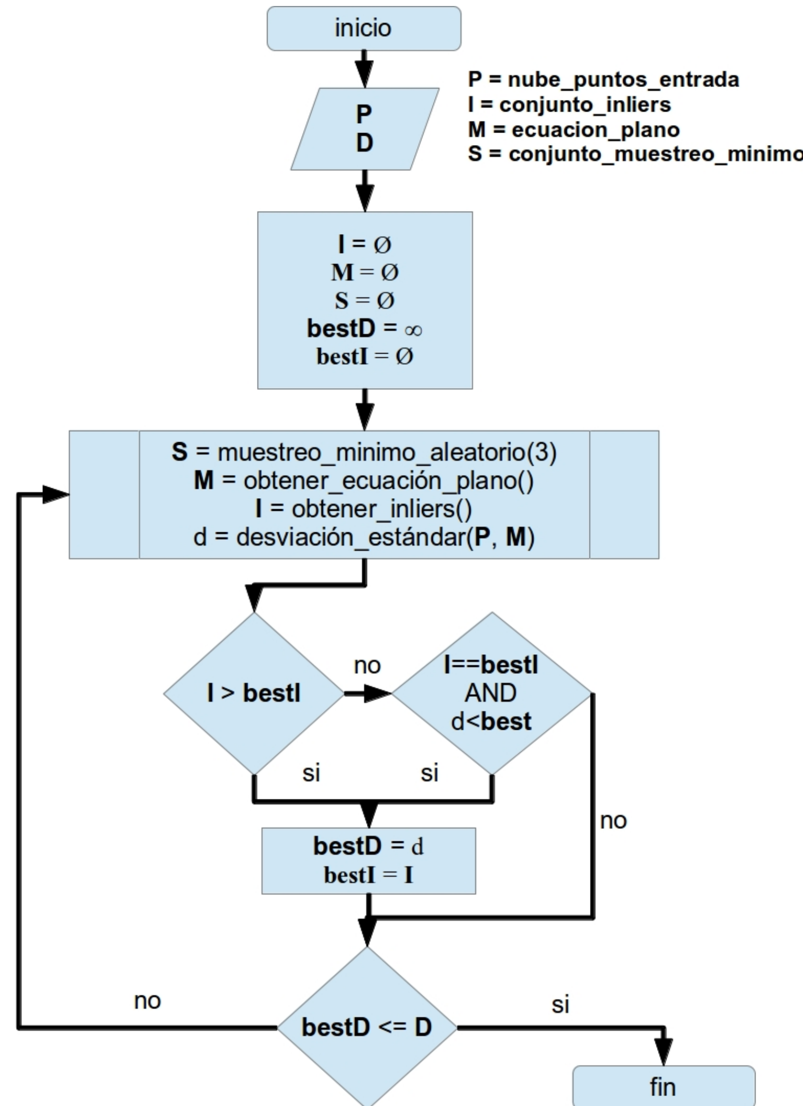

Figura 4.4: Diagrama de flujo del algoritmo RANSAC en una nube de puntos.

A continuación se detalla el flujo del proceso descrito mediante el diagrama de la Figura 4.4:

1. Se realiza la adquisición de la nube de puntos y se almacena en la variable **P**. Asimismo, se realiza la adquisición del umbral mínimo de desviación estándar y se almacena en **D**.
2. Se inicializan las variables I, M, S, bestStd y bestI, para su posterior uso.

3. Se obtiene el conjunto de muestreo mínimo (necesario para calcular la ecuación de un plano) y se almacena en **S**. Este conjunto se obtiene seleccionando, de manera aleatoria, tres diferentes datos del conjunto **P**.
4. Con base en el conjunto almacenado en **S**, se calcula la ecuación de un plano. Para esto se realiza el siguiente procedimiento:
   1. Se obtienen dos vectores A y B con base en los datos del conjunto **S**.
   2. Se realiza un producto cruz entre los vectores hallados. Mediante este producto cruz se obtiene un vector normal (N) a los vectores A y B.
   3. Las componentes del vector N son los tres primeros parámetros de la ecuación del plano, el cuarto parámetro se calcula mediante la fórmula:

$$-(Nx*S_k x+Ny*S_k y+Nz*S_k z)$$

donde: S_k es cualquier elemento del conjunto **S**.

Nx, Ny, Nz son las componentes del vector normal N.

5. Una vez obtenido la ecuación del plano, se procede a buscar todos aquellos puntos que empaten con dicha ecuación (inliers). Este proceso se encuentra ilustrado en el diagrama de la Figura 4.5:
   1. Se reciben las variables **I** (conjunto de inliers), **P** (nube de puntos) y **M** (ecuación del plano) provenientes del proceso principal.
   2. Se realiza un recorrido fila-columna sobre la nube de puntos **P**.
   3. Para cada punto en **P**:
      - Se calcula la distancia euclídea desde el plano, descrito por **M**, hacia el punto. Esto se realiza mediante la subrutina **distancia_euclidea**, que recibe como parámetros un punto 3D y la ecuación de un plano; y retorna el valor de la distancia del punto al plano.
      - Se calcula la orientación del punto respecto al plano. La subrutina **orientacion** realiza esta tarea. Esta subrutina recibe como parámetros un punto y la ecuación de un plano; y retorna ***true*** si el plano y el punto tienen la misma orientación y ***false*** en otro caso.
      - Si el punto se encuentra cerca del plano (a una distancia menor que un umbral dado **D**) y ambos se encuentran orientados en la misma dirección el punto se agrega a **I**. En caso contrario el punto no se agrega a **I**.
6. Se calcula la desviación estándar del conjunto **P** respecto a la ecuación del plano **M**. Para esto, es necesario medir la distancia euclídea desde cada punto de **P** a la ecuación del plano **M**.
7. En caso de hallar un conjunto **I** de mayor tamaño al de iteraciones anteriores y con menor desviación estándar. Se almacena la ecuación **M** calculada y el conjunto de inliers **I**.
8. Se repite el proceso desde el paso 3 hasta hallar una ecuación cuya desviación estándar respecto a **P** sea mínima.

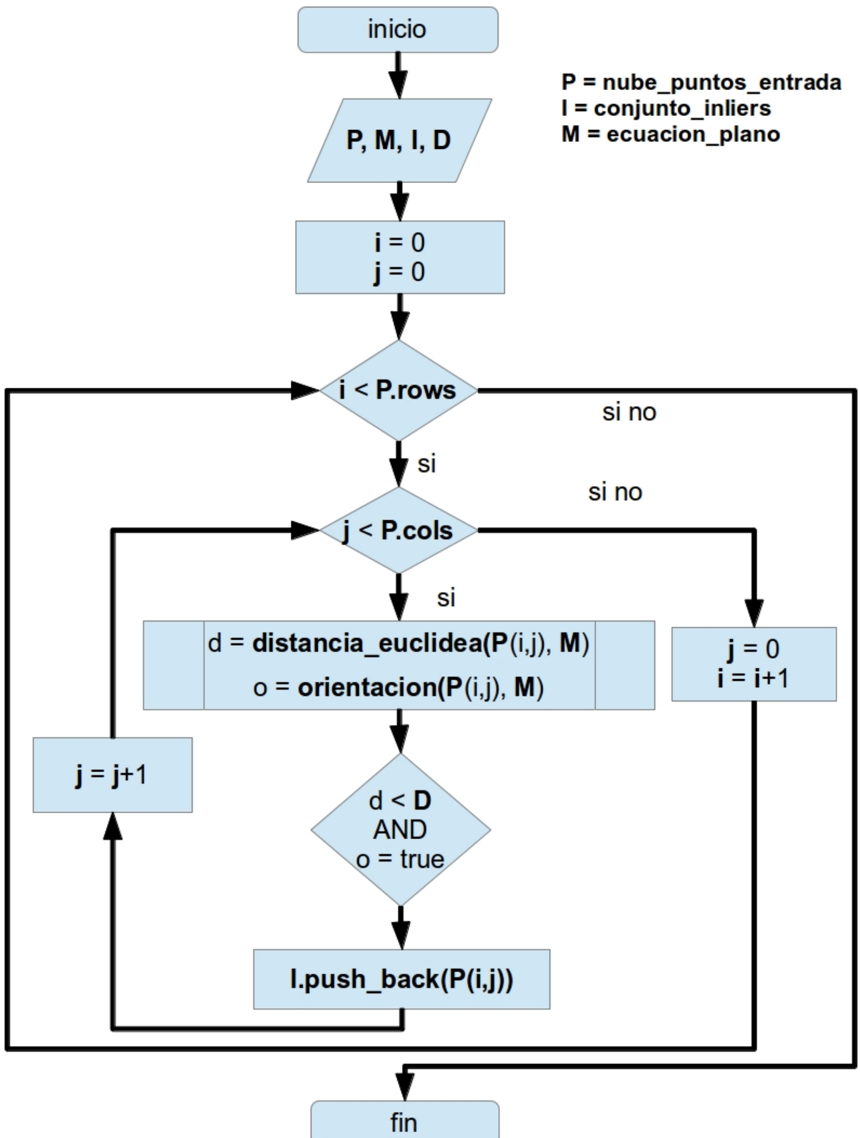

Figura 4.5: Diagrama que representa el flujo de proceso del cálculo de inliers.

Capítulo 5. Esquema de procesamiento en FPGA

Llevar a cabo el procesamiento de una nube de puntos en un FPGA trae consigo un reto importante: almacenar la nube de puntos en el FPGA. Esto debido a que una nube de puntos (adquirida mediante una cámara RGB-D de 640x480 píxeles de resolución) puede alcanzar un tamaño de hasta 7 Mega-Bytes, mientras que el FPGA Altera Cyclone (utilizado en este trabajo) permite almacenar como máximo 1 Mega-Byte de datos.

El esquema de procesamiento, propuesto en este apartado, consiste en hacer cómputo heterogéneo mediante CPU y FPGA para procesar nubes de puntos. El CPU será el encargado de realizar la adquisición de datos provenientes de la cámara RGB-D (debido a que cuenta con espacio suficiente en memoria para almacenar una nube de puntos) y sub-muestrearlos para posteriormente enviarlos al FPGA. Por su parte, el FPGA será el encargado de procesar los datos sub-muestreados y transferir al CPU los resultados de dicho procesamiento (vectores normales, ecuación del plano). En la Figura 5.1 se muestra el esquema general de procesamiento descrito anteriormente.

Mediante el sub-muestreo de datos es posible reducir una nube de puntos de 7 Mega-Bytes a un tamaño menor de 1 Mega-Byte, este sub-muestreo será realizado de la siguiente manera:

- Se reducirá la resolución de la nube de puntos, de 640x480 píxeles (VGA) a 320x240 píxeles (QVGA) y 160x120 píxeles (QQVGA).
- La precisión de datos para cada coordenada (*x,y,z*) de la nube será reducida de 64 bits a 12 bits por coordenada, para QVGA, y a 15 bits por coordenada, para QQVGA.

Llevando a cabo el sub-muestreo anteriormente descrito, se obtendrán:

- Para QVGA: 320*240 = 76800 puntos de la nube. Debido a que cada punto cuenta con tres coordenadas (*x,y,z*) se tienen 76800*3 = 230400 datos, y de acuerdo al sub-muestreo aplicado cada dato contará con una longitud de 12 bits, por lo que se tendrá un total de 230400*12 = 2764800 bits, lo cual equivale a 0.3296 Mega-Bytes (32.95% de la capacidad de almacenamiento del FPGA).
- Para QQVGA: 160*120 = 19200 (puntos) * 3 = 57600 (datos) * 15 (bits de longitud por dato) = 864000 bits. Se obtiene un total de 0.103 Mega-Bytes (10.3% de la capacidad de almacenamiento del FPGA).

La transferencia de datos del CPU al FPGA se llevará a cabo a través del puerto PCI Express mediante un protocolo de comunicación de alto nivel, descrito en el Apéndice A. Posterior a la transferencia de datos, se procederá a ejecutar las rutinas de procesamiento necesarias (calculo de vectores normales y segmentación de la superficie plana dominante) en el FPGA. Para lograr esto, deberán ser diseñados distintos módulos de hardware específico, mediante un HDL, y programados en el FPGA.

En este apartado se describen a detalle los módulos de hardware específico diseñados (para el almacenamiento y procesamiento de nubes de puntos) haciendo uso de dos herramientas:

- El entorno de desarrollo Quartus 13 en conjunto con el lenguaje de descripción de hardware VHDL (para llevar a cabo la descripción de hardware y simulaciones).
- La tarjeta de desarrollo Intel DE2i-150 (para la ejecución y pruebas del hardware desarrollado).

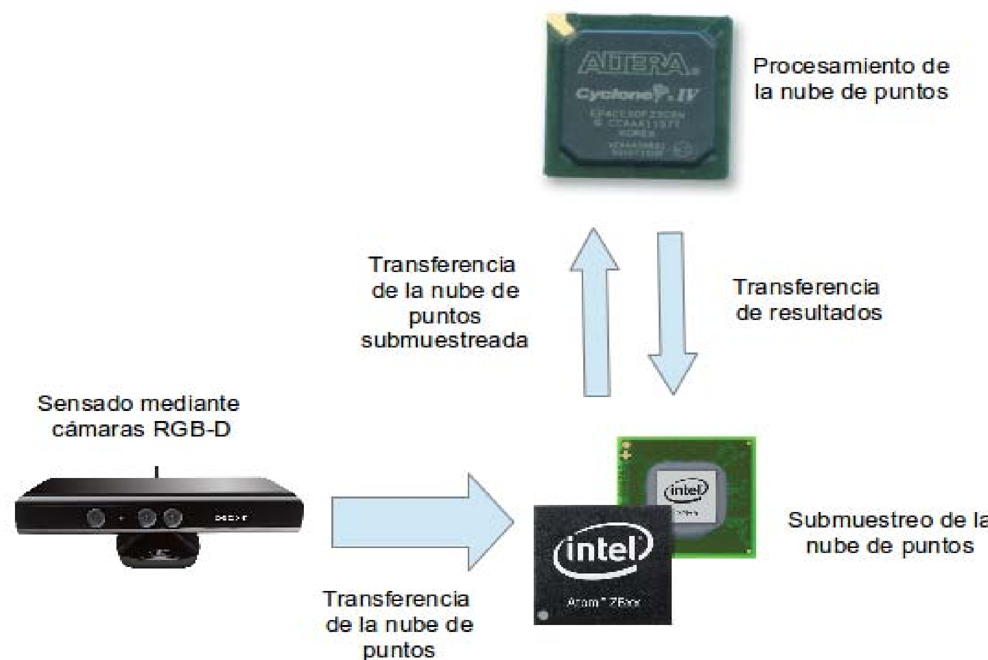

Figura 5.1: Esquema de procesamiento de nubes de puntos mediante FPGA y CPU.

5.1. Sistema de memoria

En procesamiento de nube de puntos, los datos almacenados en memoria juegan un papel importante dado que, la memoria es el lugar en el que se almacenan:

- La nube de puntos.
  Los datos resultantes de cálculos intermedios
- Los resultados del procesamiento.

Como ya se mencionó anteriormente, la nube de puntos entregada por una cámara RGB-D se almacenan en formato *fila*,*columna*. Esto permite organizar la nube como un arreglo 2D de datos (tal como se encuentra organizada una imagen RGB) y almacenarla en la memoria del FPGA. En [DV12] se presenta un esquema de organización de memorias para almacenar imágenes 2D en FPGA el cual reduce el número de accesos a memoria al momento de hacer lecturas. El sistema de memoria desarrollado en este trabajo, detallado a continuación, se encuentra basado en este esquema.

Gracias a que la nube de puntos se encuentra organizada mediante sus valores de *fila* y *columna*, es posible dividirla en cuatro conjuntos de datos diferentes y almacenar cada uno de estos conjuntos en un bloque de memoria del FPGA. Esta división se realiza con base en la paridad de los valores de *fila* y *columna* como se detalla a continuación:

- En el primer bloque de memoria (B00) se almacenará el conjunto de puntos cuyos valores de fila y columna sean número pares.
- En el segundo bloque (B01) se almacenará el conjunto de puntos cuyo valor de fila sea un número par y valor de columna sea un número impar.
- En el tercer bloque (B10) se almacenará el conjunto de puntos cuyo valor de fila sea un número impar y valor de columna sea un número par.
- En el cuarto bloque (B11) se almacenará el conjunto de puntos cuyos valores de fila y columna sean números impares.

En la Figura 5.2 se muestra la división de un arreglo 2D utilizando esta técnica.

Mediante esta organización es posible obtener cuatro elementos contiguos de la nube de puntos con un solo ciclo de acceso a memoria Figura 5.3. Lo anterior es posible gracias a que los datos se almacenan en cuatro bloques de memoria independientes. En la Figura 5.4 se muestra el diagrama de conexiones para el manejo de los cuatro bloques de memoria en los cuales se almacenan secciones diferentes de la nube de puntos. Su funcionamiento se detalla a continuación:

- Los bloques B00, B01, B10 y B11 consisten en memorias RAM (cada localidad corresponde a un punto de la nube de puntos).
- Cada una de estas memorias RAM cuenta con operaciones de lectura y escritura simples (un solo dato y dirección por ciclo).

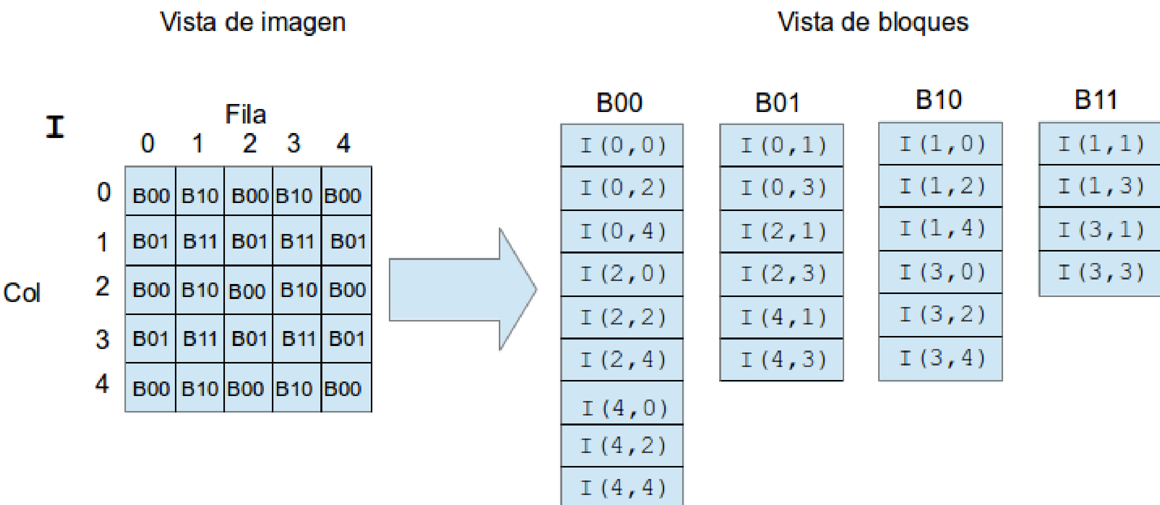

Figura 5.2: Esquema de división de memoria por bloques. Del lado izquierdo se muestra cómo se organizan las nubes de puntos en el CPU, del lado derecho la organización por bloques propuesta.

Direcciones de entrada: (fila,columna) = (2,1)

fila=0 columna=0	fila=0 columna=1	fila=0 columna=2	fila=0 columna=3
fila=1 columna=0	fila=1 columna=1	fila=1 columna=2	fila=1 columna=3
fila=2 columna=0	fila=2 columna=1	fila=2 columna=2	fila=2 columna=3
fila=3 columna=0	fila=3 columna=1	fila=3 columna=2	fila=3 columna=3

Figura 5.3: Interpretación gráfica de los datos obtenidos de un arreglo 2D en un ciclo de lectura.

- El módulo ADDR_GEN traduce las direcciones de datos de *fila*,*columna* a direcciones de memoria para cada bloque RAM, para esto hace uso de la paridad de los valores de *fila* y *columna*.
- La escritura en cada una de las memorias RAM es controlada mediante un multiplexor (MUX) que activa la señal de escritura en el bloque correspondiente a la paridad de los valores de *fila* y *columna*:
    - Si *fila* y *columna* son valores pares entonces se activa B00 para escritura.
    - Con valores de *fila* y *columna* impares se activa B11.
    - Para un valor de *fila* par y un valor de *columna* impar, el bloque B01 será activado.
    - Para un valor de *fila* impar y un valor de columna *par*, se activará el bloque B10.

Cabe destacar que, por cada ciclo de escritura, solo es posible escribir un dato en el bloque apuntado por *fila* y *columna*.

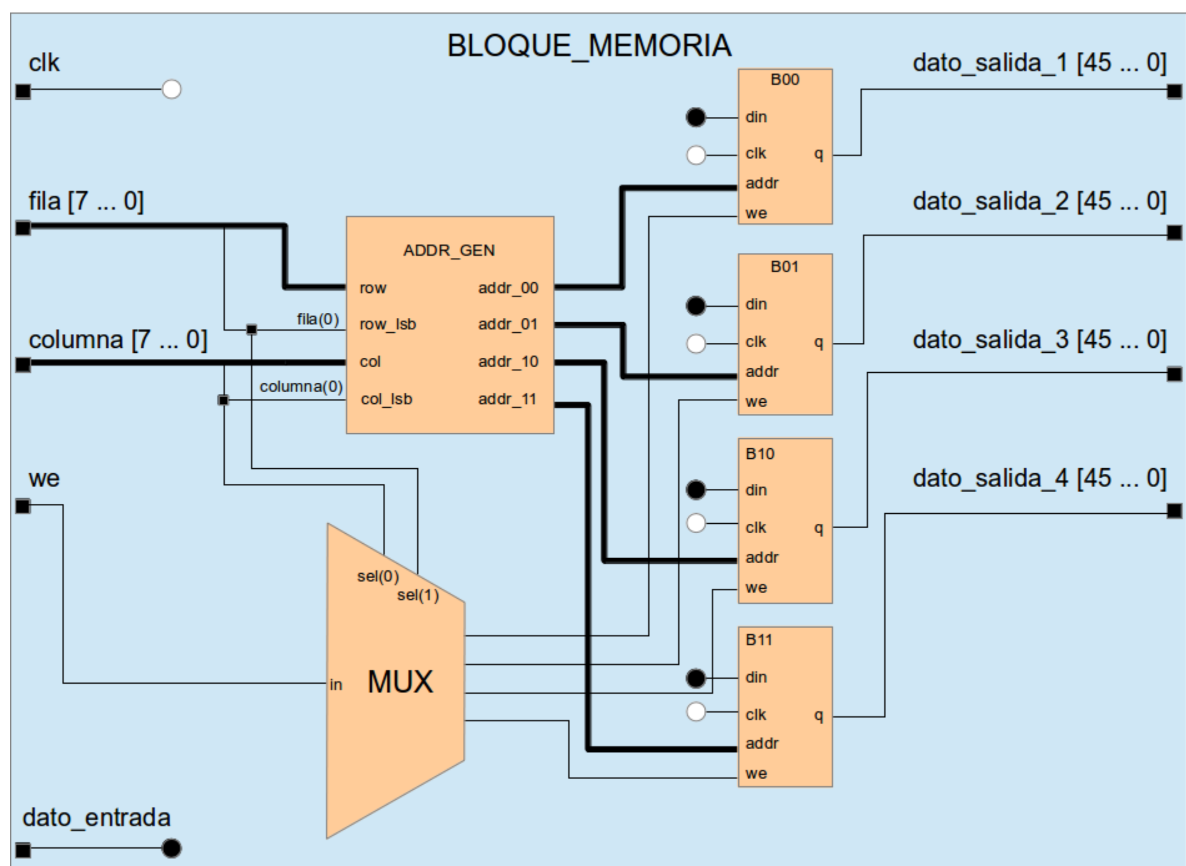

Figura 5.4: Memoria de nube de puntos.

5.2. Módulo de extracción de puntos característicos

En el Algoritmo 3.1 se describe como realizar el cálculo de vectores normales (puntos característicos) sobre una nube de puntos. Dicho algoritmo describe el problema de forma secuencial (una instrucción detrás de otra) por lo que, resulta relativamente sencillo realizar la traducción desde el algoritmo hacia un lenguaje de computadora de alto nivel (dado que la mayoría de los programas se ejecutan de forma secuencial). Sin embargo, realizar la traducción de un algoritmo secuencial a un circuito electrónico puede volverse una tarea complicada (dado que las señales e instrucciones no se ejecutan de forma secuencial en un circuito electrónico). Por tal motivo, es tarea del diseñador crear la arquitectura de hardware necesaria para ejecutar una serie de instrucciones de manera secuencial mediante un circuito electrónico. A continuación se presenta la arquitectura de hardware diseñada para ejecutar el Algoritmo 3.1 en un FPGA.

En la Figura 5.5 se muestra el diagrama, en alto nivel, del módulo de hardware diseñado para el cálculo de vectores normales en una nube de puntos. El módulo cuenta distintos componentes:

- Memorias de datos (***memoria de nube de puntos*** y ***memoria de vectores normales***).
- Registros (***A***, ***B***, ***fila***, ***col***). Se encargan de almacenar los resultados intermedios requeridos para el cálculo de los vectores normales.
- Módulos de procesamiento especializados (***producto cruz*** y ***generador fila-columna***).
- Unidad de control: Encargada de generar las señales de control necesarias para el correcto funcionamiento del sistema.

5.2.1. Memoria de nube de puntos y memoria de vectores normales.

Ambos bloques de memoria son instancias del sistema descrito en el apartado 5.1. La ***memoria de nube de puntos*** contiene la nube de puntos proveniente del CPU. Por otro lado la ***memoria de vectores normales*** almacena todos los vectores normales obtenidos como resultado de procesar la nube de puntos completa. Una vez concluido el procesamiento, todos los datos contenidos en la ***memoria de vectores normales*** son enviados de vuelta a la CPU.

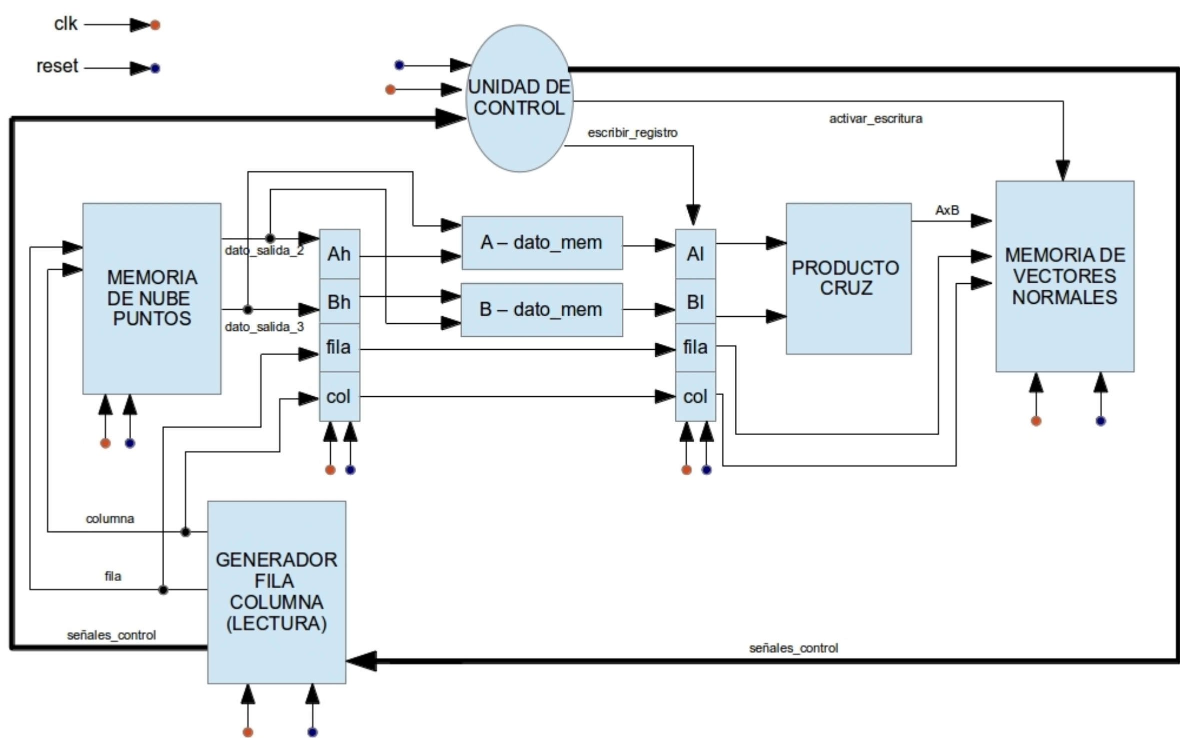

Figura 5.5: Diagrama de hardware de alto nivel para el proceso de extracción de puntos característicos de una nube de puntos.

5.2.2. Registros A, B, fila y col.

Los registros ***A*** y ***B*** almacenarán los vectores A y B (ver algoritmo 2.1), mientras que ***fila*** y ***col*** contendrán los valores de fila y columna correspondientes a la dirección de memoria en la cual se guardará el vector normal a A y B.

5.2.3. Módulo producto-cruz.

El módulo ***producto cruz*** es un componente combinacional que se encarga de calcular el producto cruz entre dos vectores (almacenados en los registros ***A*** y ***B***). La Figura 5.6 muestra el diagrama de conexiones internas de este módulo. Su funcionamiento es el siguiente:

- Cada vector (*vector_A* y *vector_B*) cuenta con una longitud de 45 bits, 15 bits por cada coordenada *x,y,z* del vector (la Tabla 5.1 muestra la correspondencia entre las coordenadas de un vector y su ubicación en el arreglo de bits).
- Al realizar la multiplicación de dos coordenadas se obtiene un nuevo dato de 30 bits de longitud, el cual, será sumado con otro dato de 30 bits de longitud.

- El resultado de cada sumador es truncado y almacenado en el *vector_n* el cual contendrá el vector normal a *vector_A* y *vector_B*.

Dado que este módulo es completamente combinacional, es posible obtener el producto cruz de dos vectores en un único ciclo de reloj.

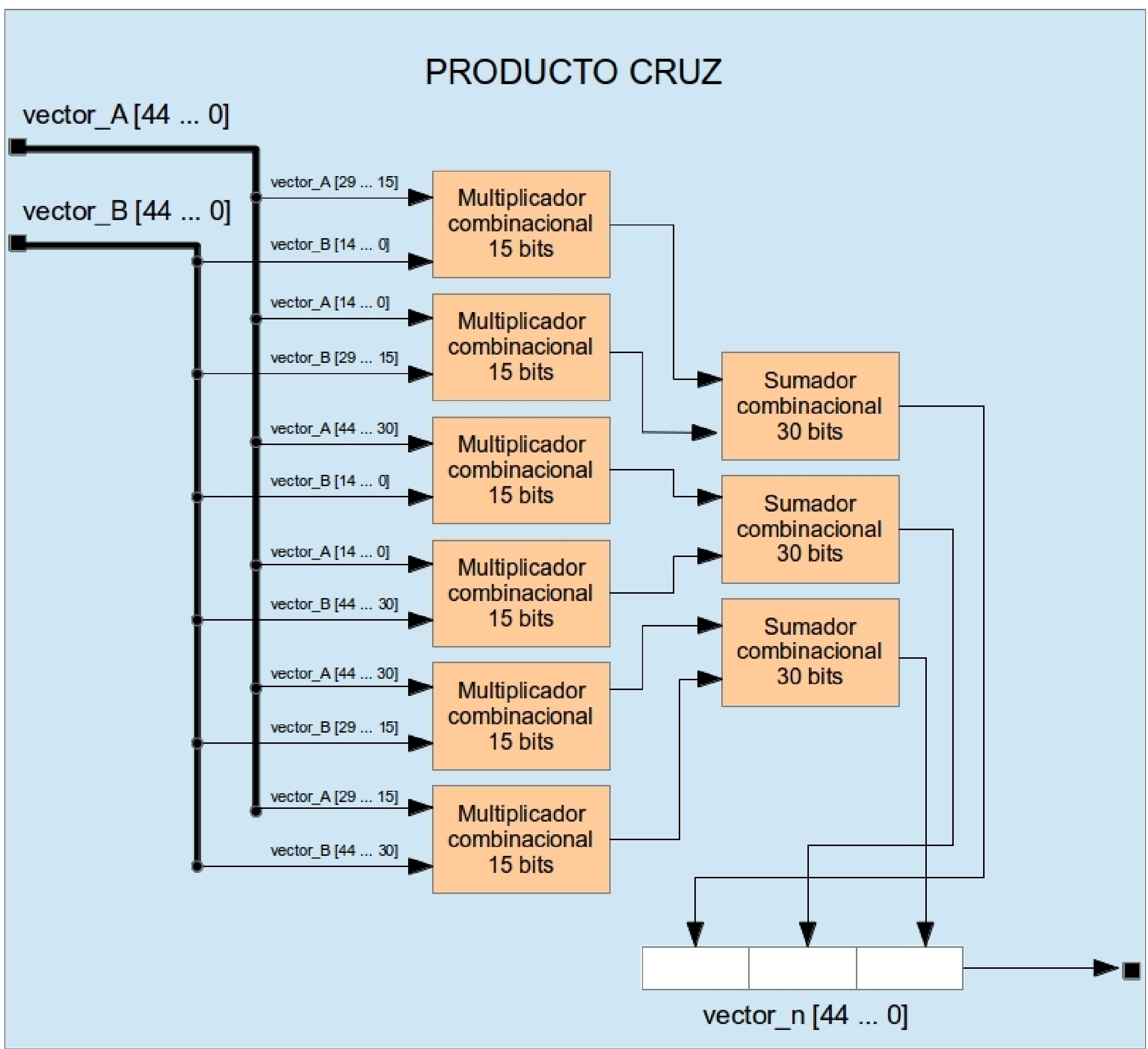

Figura 5.6: Conexiones internas el módulo producto cruz.

Coordenada	Correspondencia en bits
Vector(x)	*Vector [44 ... 30]*
Vector(y)	*Vector [29 ... 15]*
Vector(z)	*Vector [14 ... 0]*

Tabla 5.1: Tabla de correspondencia entre coordenadas espaciales y posición en bits.

5.2.4. Módulo generador fila-columna.

El ***generador fila-columna*** es el módulo encargado de, como su nombre lo indica, genera los valores de fila y columna para realizar un recorrido por la nube de puntos. La Figura 5.7 muestra el diagrama esquemático del generador. Los valores almacenados en los registros *fila* y *columna* del generador actualizan su valor con base en las señales provenientes de la unidad de control (*incrementar_fila*, *incrementar_columna*, *incrementar_fila_columna* y *decrementar_fila_columna*) así mismo, cada vez que se actualiza el valor de cualquier registro, se generan señales de control (*fila_incrementada*, *columna_incrementada*) indicando que un registro ha sido actualizado.

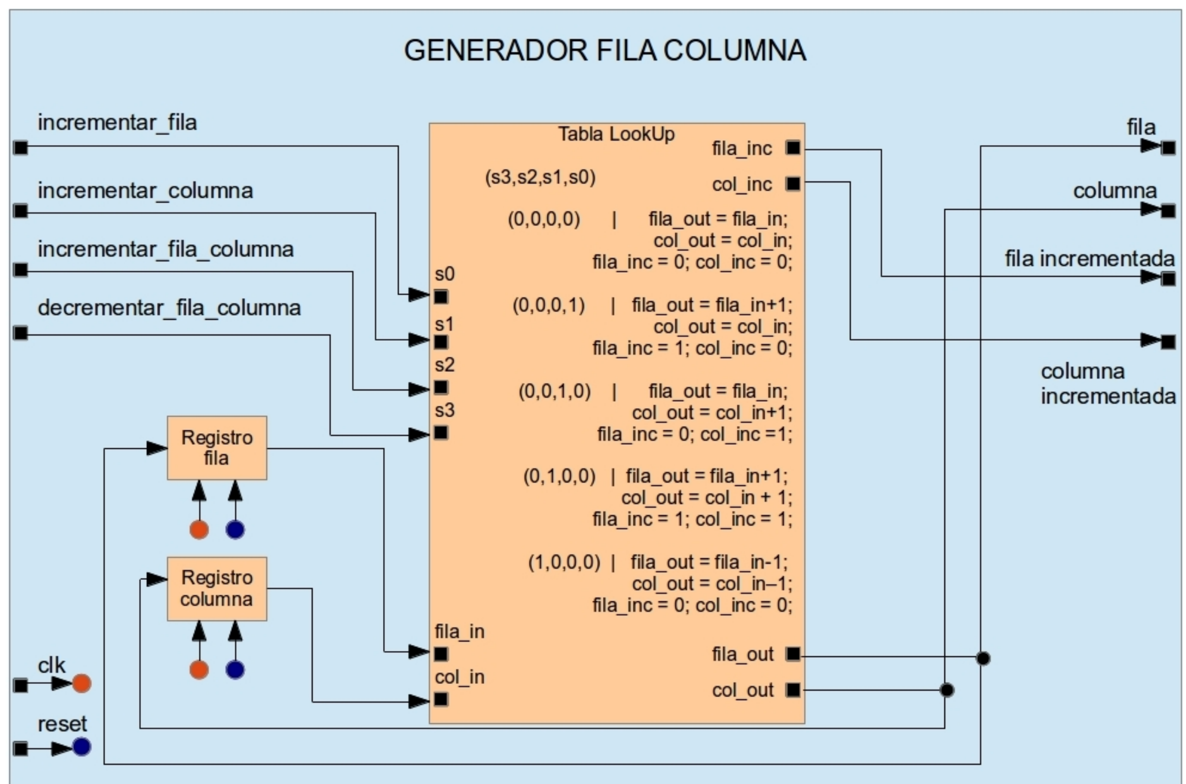

Figura 5.7: Diagrama de alto nivel del módulo generador de fila y columna.

5.2.5. Unidad de control.

Este módulo se encarga de generar todas las señales de control necesarias para llevar a cabo el cálculo de los vectores normales sobre la nube de puntos. En la Figura 5.8 se muestra en forma de carta ASM el funcionamiento de la ***unidad de control***. A continuación se describe la ejecución de la máquina de estados:

- *stdBy*: En este estado se inicializan con ceros los valores de todos los registros, esto provoca que las direcciones de entrada (fila,columna) de la memoria de nube de puntos sean también ceros. A este estado se accede cuando la señal *reset* se encuentra activa.
- *s0*: A este estado se accede a través de los estados *stdby* o *s2*. En este estado, se obtienen dos puntos, de la memoria de nube de puntos, necesarios para hacer el cálculo del vector normal (estos puntos se almacenarán en los registros ***Ah*** y ***Bh***). Así mismo se activa la señal *incrementar_fila_columna* para obtener (en el siguiente estado) los siguientes dos puntos necesarios para calcular el vector normal.
- *s1*: Se obtienen, en la salida de la memoria de nube de puntos, los siguientes dos puntos requeridos para el cálculo del vector normal. Se realiza una resta entre los puntos obtenidos en este estado y los puntos almacenados en los registros ***Ah*** y ***Bh***. De esta forma se obtienen dos vectores tangenciales a una superficie, cada vector será almacenado en su registro correspondiente (***Al*** o ***Bl***) en el siguiente estado (para esto es necesario activar la señal *escribir_registro*. Así mismo, se activa la señal *decrementar_fila_columna* para regresar a la posición original del recorrido de la nube de puntos.
- *s2*: En este estado, se lee el valor de los registros ***Al*** y ***Bl*** (que contienen dos vectores tangenciales a una superficie) y se realiza el producto cruz entre ambos (el resultado se almacenará en la memoria de vectores normales) mediante el módulo combinacional ***producto cruz***. Así mismo se activan las señales *incrementar_fila* (para continuar el recorrido en la nube de puntos) y *activar_escritura* (para escribir el resultado del producto cruz en la memoria de vectores normales).

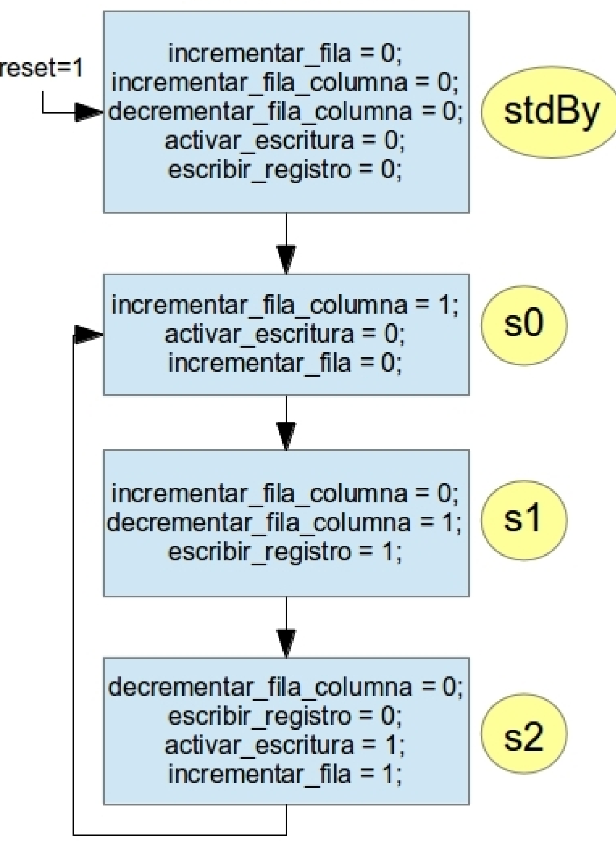

Figura 5.8: Funcionamiento de la unidad de control representado en forma de carta ASM.

Capítulo 6. Esquema de procesamiento en GPU

Como se mencionó en el capítulo 2, la arquitectura interna de una GPU consiste de múltiples núcleos (procesadores de propósito general) controlados por la misma unidad de control (arquitectura SIMD). Esto es, todos los procesadores de la GPU llevan a cabo el mismo procesamiento sobre distintos bloques de datos al mismo tiempo. Lo anterior, es un tipo de ejecución en paralelo y permite que una GPU pueda procesar un gran volumen de datos en un tiempo relativamente corto (teóricamente, lo que tarda el procesamiento en un sólo núcleo de la GPU). Para aprovechar el paralelismo masivo que ofrecen las GPU, en el año 2001 se popularizó el concepto conocido como Programación de Propósito General para GPU (GPGPU por sus siglas en inglés).

GPGPU se refiere al uso de paralelismo, que ofrecen las GPU, para la optimización de procesos que ejecuta un CPU de manera secuencial. Para implementar una aplicación utilizando el concepto GPGPU se debe contar con, al menos, dos arquitecturas de hardware distintas (un CPU y una GPU) y realizar el intercambio de información entre ambas arquitecturas de forma ordenada. Así mismo, se debe contar con una plataforma de desarrollo que permita escribir código específico para ser ejecutado en cada una de las arquitecturas. Existen distintas plataformas de desarrollo que permiten escribir y compilar código específico para CPU y GPU. La opción mas extendida en la actualidad es la plataforma CUDA, desarrollada por la compañía nVidia.

Con el surgimiento de GPGPU, los desarrolladores de software se han visto a la tarea de implementar aplicaciones para su ejecución en dos modelos distintos. Por un lado se tiene el modelo secuencial, en el cual las instrucciones son ejecutadas una a la vez de manera concurrente por un mismo procesador. Por el otro lado se tiene el modelo de ejecución en paralelo, donde distintas instrucciones son ejecutadas al mismo tiempo por distintos procesadores. El desarrollo de aplicaciones para el modelo de ejecución secuencial consiste en escribir código para el CPU, mientras que, para el modelo de ejecución en paralelo, se trata de escribir código para la GPU. Es necesario, entonces, que el desarrollador domine ambos paradigmas.

En este capítulo se detalla la implementación de aplicaciones mediante GPGPU correspondientes a los algoritmos 3.1 (Extracción de Puntos Característicos) y 3.2 (Segmentación de la Superficie Plana Dominante). Así mismo se describen las distintas configuraciones de hardware para un sistema heterogéneo CPU+GPU y el modelo de programación que un desarrollador debe seguir con CUDA.

6.1. Arquitectura de hardware heterogéneo (CPU+GPU)

Para abordar el desarrollo de software en un sistema de cómputo heterogéneo, conformado por las arquitecturas CPU y GPU, es necesario: entender como funcionan ambas arquitecturas y conocer la configuración necesaria para que ambas interactúen entre sí. En la sección 2.1 se describió la organización interna y el funcionamiento de ambas arquitecturas de manera individual, sin embargo, no se presentó la configuración de un sistema heterogéneo CPU+GPU ni se describió la forma en que estas dos arquitecturas se comunican y comparten datos. En este apartado se presentan distintas configuraciones, a nivel de hardware, para un sistema heterogéneo CPU+GPU.

6.1.1. Configuración básica CPU/GPU

En la Figura 6.1 se muestra, de forma simplificada, la configuración de un sistema CPU+GPU. Cada una de las arquitecturas involucradas en esta configuración cuenta con su propio sistema de memoria y comparten datos a través de una interfaz de comunicación.

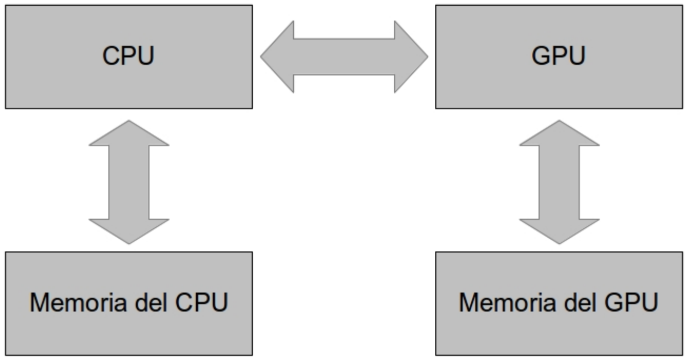

Figura 6.1: Arquitectura simplificada de un sistema CPU+GPU.

En un CPU, la comunicación con los periféricos se realiza a través de un componente del CPU por el cual transita cada dato que sale del CPU al exterior: el “chipset”. Por lo tanto cada bit que transita del CPU al GPU pasa a través de este componente. En los CPU actuales, el chipset se encuentra dividido en dos integrados: el “puente sur” (southbridge) que conecta al CPU con la mayoría de los periféricos y el “puente norte” (northbridge) mediante el cual se conecta el CPU con el bus gráfico (actualmente PCI Express) y con la memoria RAM. La Figura 6.2 muestra una versión más detallada de la configuración del sistema CPU+GPU, incluyendo el chipset northbridge y el bus gráfico PCI Express.

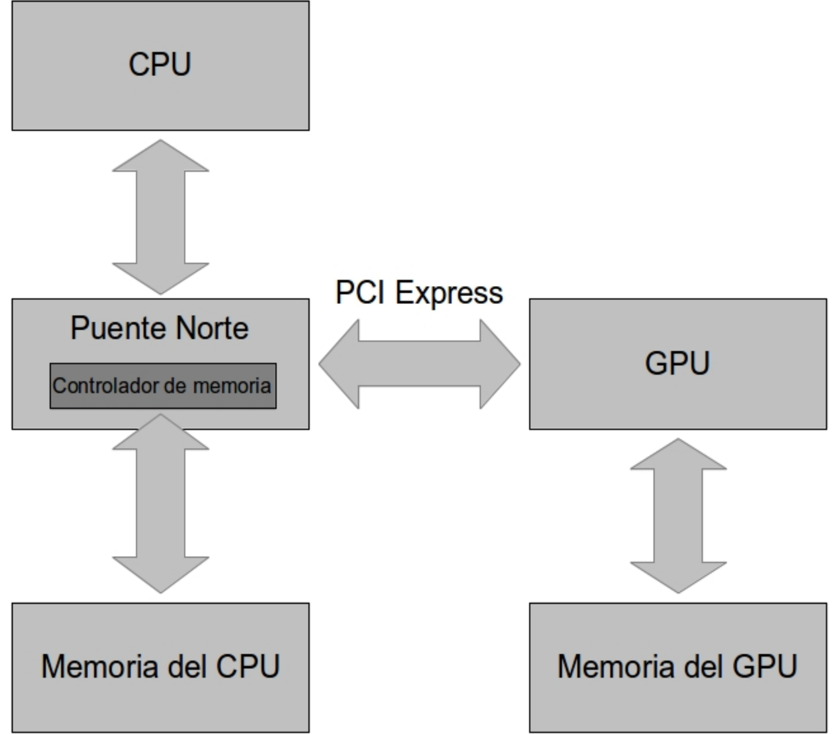

Figura 6.2: Sistema heterogéneo CPU+GPU conectada mediante el "puente norte".

6.1.2. Configuración CPU/GPU con múltiples CPU

Los diagramas revisados anteriormente nos muestran configuraciones CPU+GPU en los cuales existe solamente un único CPU y un único GPU. Sin embargo, en las computadoras actuales se cuenta con múltiples procesadores en un solo chip y, en ocasiones, con más de un chip de procesamiento gráfico. Esto resulta en sistemas CPU+GPU que pueden llegar a contar con múltiples CPU y múltiples GPU. En este apartado se revisa la configuración multi-CPU.

La Figura 6.3 muestra una configuración CPU+GPU con múltiples CPU. Se puede notar que, a diferencia de la Figura 6.2, cada CPU se conecta directamente a la memoria. Lo anterior es debido a que en procesadores actuales (a partir de AMD Opteron y de la familia Core i7 de Intel) el controlador de memoria se encuentra integrado dentro de cada procesador. Así mismo, se puede observar que cada CPU cuenta con su propio banco de memoria y que estos se encuentran conectados en un enlace punto a punto (a esta conexión punto a punto se le conoce como HyperTransport en AMD y como QuickPath Interconnect en Intel). HyperTransport y QuickPath Interconnect permiten que cada CPU tenga acceso a los datos en memoria de cualquier otro CPU.

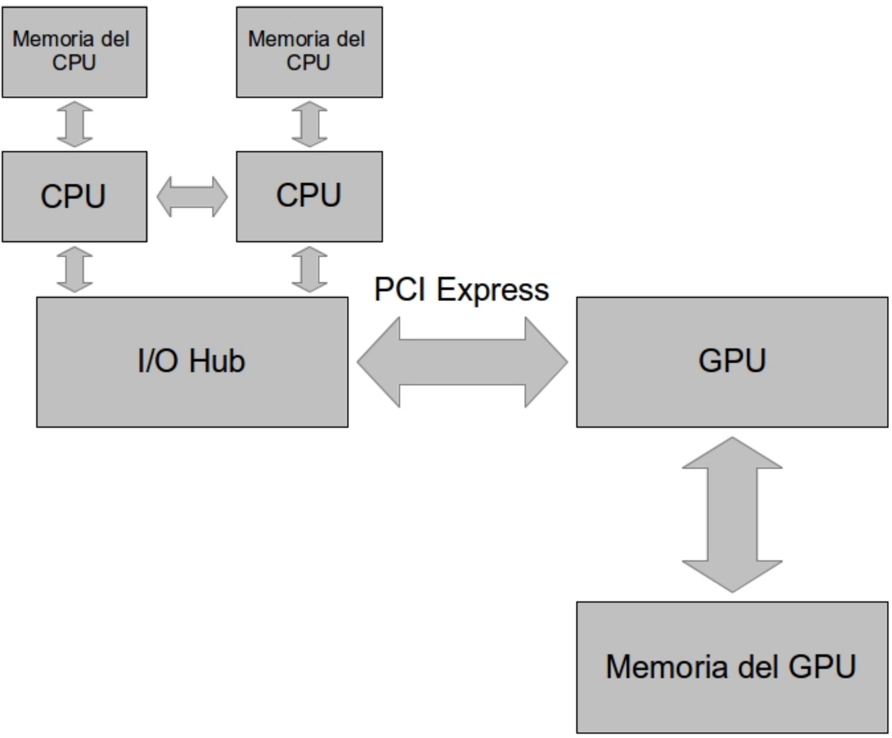

Figura 6.3: Configuración de un sistema heterogéneo CPU+GPU con múltiples CPU.

En este tipo de sistema multi CPU, la comunicación con el GPU, y los periféricos, se realiza a través de un Hub de Entrada Salida (I/O Hub). Este hub maneja la comunicación, via PCI Express, entre los múltiples CPU y el GPU. Como se menciona en [Wilt13], CUDA trabaja muy bien con este tipo de configuración.

6.1.3. Configuración CPU/GPU con múltiples GPU

Para los sistemas que involucran múltiples GPU, existen distintas alternativas de configuración, la Figura 6.4 muestra una de ellas. En este tipo de configuración dos GPU se encuentran conectadas al northbridge del CPU mediante el puerto PCI Express. Cada GPU cuenta con su propio sistema de memoria y se encuentran conectadas en puertos PCI Express individuales. Para que dos GPU pudieran coexistir en la misma placa base en 2004 la compañía nVidia introdujo la tecnología de Interfaz de Enlace Escalable (SLI por sus siglas en inglés), esta tecnología permite que dos GPU puedan trabajar en paralelo en la misma tarjeta madre. Para el usuario final, la tecnología SLI de nVidia permite trabajar con esta organización como si se tratase de la presentada en la Figura 6.2.

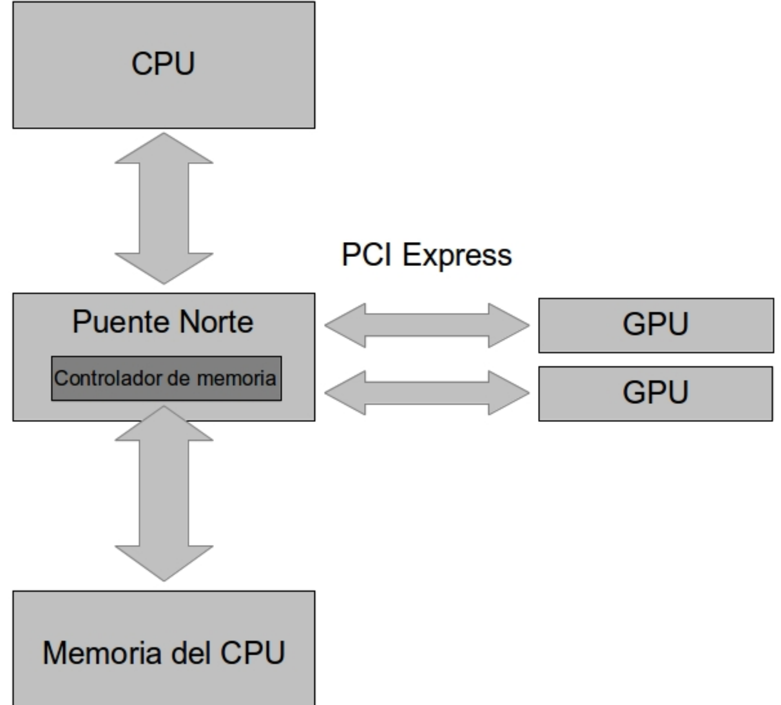

Figura 6.4: Configuración de un sistema heterogéneo CPU+GPU con múltiples GPU conectados al northbridge.

Otro tipo de configuración con múltiples GPU consiste en encapsular dos GPU en una única tarjeta (Dual-GPU) y conectar esta al CPU (Figura 6.5). En este tipo de configuración cada GPU cuenta con su propio sistema de memoria (es decir, no comparten datos de memoria entre ellas) y se comunican a través de PCI Express. Mediante SLI es posible trabajar con esta organización como si se tratase de un solo GPU. Sin embargo, para el desarrollo de aplicaciones, SLI no hace esta organización transparente al desarrollador, se debe deshabilitar SLI para desarrollar software para esta arquitectura.

Actualmente, gracias a las prestaciones de los CPU, placas base y GPU actuales, es posible conectar múltiples tarjetas Dual-GPU con sistemas multi CPU. Este tipo de configuración permite un alto poder de cómputo a coste de elevar la complejidad del desarrollo. La Figura 6.6 muestra un ejemplo de esta configuración.

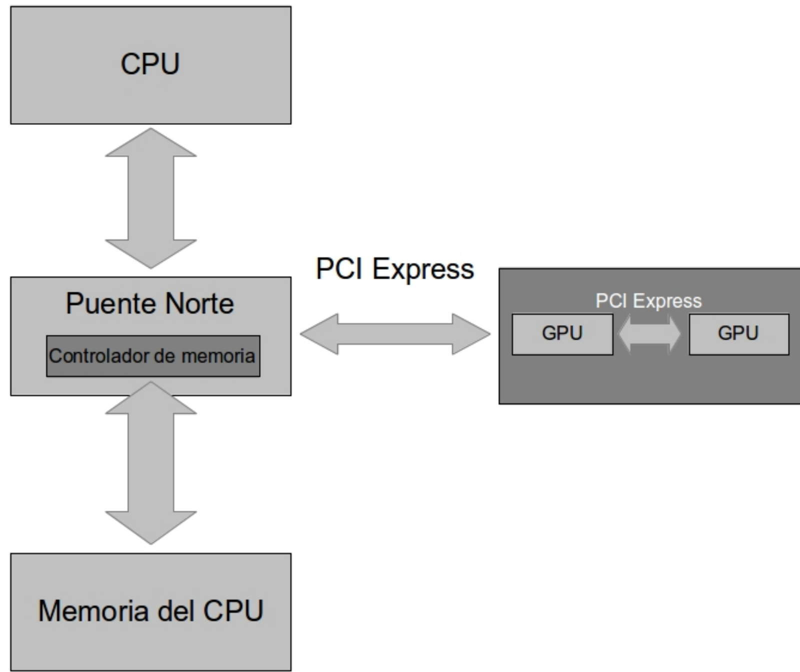

Figura 6.5: Configuración de un sistema heterogéneo CPU+GPU con múltiples GPU encapsulados en un único chip.

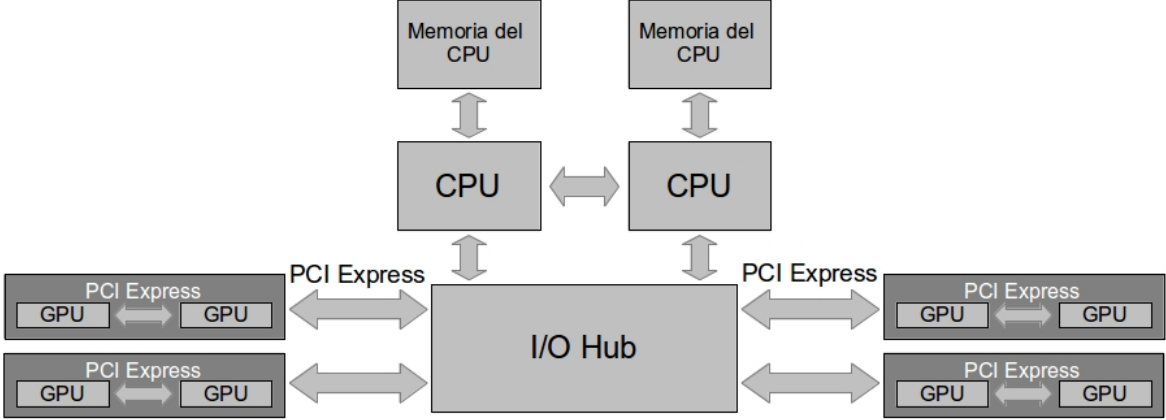

Figura 6.6: Configuración de un sistema heterogéneo CPU+GPU con múltiples CPU y múltiples chips Dual-GPU.

6.2. Arquitectura de software CUDA

Escribir código para una arquitectura específica implica contar con un lenguaje de programación diseñado para tal propósito, además de conocer a fondo el funcionamiento de dicha arquitectura. Entonces, para desarrollar aplicaciones en un sistema de cómputo heterogéneo, se debe contar con lenguajes de programación especializados para cada arquitectura involucrada. Además, es necesario crear interfaces de software para sincronizar las interacciones entre las arquitectura que conforman al sistema.

Actualmente, para los sistemas CPU+GPU, existen distintas plataformas de desarrollo que: por un lado proveen lenguajes de programación especializados para cada arquitectura y, por el otro lado, se encargan de sincronizar las interacciones entre ambas arquitecturas (intercambio de datos, control de ejecución, etc.). Una de las plataformas más populares actualmente es CUDA. CUDA fue creada por la compañía nVidia y permite desarrollar aplicaciones para sistemas CPU+GPU (siempre y cuando se cuente con un GPU de la marca nVidia).

La plataforma CUDA provee un conjunto de librerías y controladores que manejan las interacciones entre el CPU y el GPU. Así mismo provee el lenguaje de desarrollo CUDA C, que es una extensión del lenguaje de programación C. Mediante CUDA C es posible escribir código para ambas arquitecturas (CPU y GPU) y definir los segmentos de código que será ejecutados en cada una de ellas. A continuación se describirá la arquitectura general de las aplicaciones desarrolladas con CUDA C y la forma en como estas aplicaciones son ejecutadas por los sistemas CPU+GPU.

6.2.1. Estructura de un programa con CUDA C

Antes de revisar las partes que componen a una aplicación desarrollada con CUDA C, es necesario presentar la terminología relacionada al modelo de programación con CUDA C.

- *Dispositivo*: Es el nombre que se le da a cada GPU del sistema de manera individual.
- *Equipo anfitrión*: Es el nombre que se le da al CPU.
- *Kernel*: Bloque de código que será ejecutado en determinados núcleos de la GPU.
- *Hilo*: Representa el estado de ejecución (asociado al kernel) de cada núcleo de la GPU. Una GPU es capaz de tener múltiples núcleos ejecutando el kernel de forma simultanea (hilos en paralelo).
- *Bloque*: Conjunto de hilos que son ejecutados juntos en la GPU.
- *Malla*: Conjunto de bloques que se ejecutan juntos en la GPU.

Una aplicación escrita con CUDA C contiene dos componentes principales: el código escrito para el dispositivo (kernel) y el código escrito para el equipo anfitrión. En un sistema CPU+GPU, el kernel es ejecutado en múltiples núcleos del dispositivo de forma simultanea; por otro lado, el código del equipo anfitrión se ejecuta de manera concurrente en el CPU. Dado que el driver de CUDA se encuentra instalado en el sistema operativo y éste a su vez en el equipo anfitrión, suele ser éste último quien controla la ejecución de la aplicación. Por otro lado el dispositivo actúa como un coprocesador (orientado hacia el procesamiento masivo de datos).

Parte del control de la ejecución de una aplicación (las interacciones de más bajo nivel) son llevadas a cabo por el equipo anfitrión a través del driver de CUDA. Por otro lado, las interacciones de más alto nivel (como la transferencia de datos entre arquitecturas) es posible controlarlas mediante determinadas funciones provistas por las librerías que proporciona CUDA. Mediante estas funciones, el desarrollador de aplicaciones puede determinar (mediante código de CUDA C) cuando estas interacciones deben llevarse a cabo.

Una de las interacciones más utilizadas en las aplicaciones CPU+GPU es la transferencia de datos entre arquitecturas. Mediante código de CUDA C es posible controlar la transferencia de datos entre el equipo anfitrión y el dispositivo. Para esto, CUDA provee las funciones cudaMalloc y cudaMemcpy. La función cudaMalloc es utilizada para reservar espacio de memoria en el dispositivo. Por otro lado cudaMemcpy se usa para copiar datos desde el equipo anfitrión al dispositivo (y viceversa). En una aplicación es común realizar la transferencia de datos desde el equipo anfitrión hacia el dispositivo para su procesamiento, y posteriormente (una vez que termina el procesamiento de datos en el dispositivo) transferir los datos resultantes desde el dispositivo al equipo anfitrión.

La Figura 6.7 muestra el código fuente de una aplicación escrita para un sistema heterogéneo CPU+GPU. Esta aplicación contiene los elementos básicos que todo programa escrito con CUDA C debe tener:

1. Se reserva espacio de memoria (en el dispositivo y en el equipo anfitrión) para almacenar los datos a procesar.
2. Se realiza el envío de datos desde el equipo anfitrión hacia el dispositivo.
3. Se carga y ejecuta el kernel en el dispositivo.
4. Una vez terminada la ejecución del kernel en el dispositivo. Se realiza el envío de datos desde el dispositivo hacia el equipo anfitrión.
5. Se libera la memoria reservada en el paso 2.

Código escrito para el dispositivo

```
__global__ void mykernel(int *a, int *b, int *c)
{
        c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}

void main()
{
        int *arrayA, *arrayB, *arrayC, *dev_a, *dev_b, *dev_c;
        int size = sizeof(int)*ArraySize;

        //Reservar espacio en equipo anfitrion
        arrayA = (int *)malloc(size);
        arrayB = (int *)malloc(size);
        arrayC = (int *)malloc(size);

        //Inicializar los arreglos
        random_ints(arrayA, ArraySize);
        random_ints(arrayB, ArraySize);

        //Reservar espacio en dispositivo
        cudaMalloc((void **)&dev_a, size);
        cudaMalloc((void **)&dev_b, size);
        cudaMalloc((void **)&dev_c, size);

        //Enviar datos a dispositivo
        cudaMemcpy(dev_a, arrayA, size, cudaMemcpyHostToDevice);
        cudaMemcpy(dev_b, arrayB, size, cudaMemcpyHostToDevice);

        //Cargar y ejecutar el kernel
        mykernel<<<1,ArraySize>>>(dev_a, dev_b, dev_c);

        //Enviar datos de dispositivo a equipo anfitrion
        cudaMemcpy(arrayC, dev_c,size, cudaMemcpyDeviceToHost);

        //liberar la memoria del CPU
        free(arrayA);free(arrayB);free(arrayC);
        //liberar la memoria del gpu
        cudaFree(dev_a);cudaFree(dev_b);cudaFree(dev_c);
}
```

Código escrito para el equipo anfitrión

1 2 3 4 5

Figura 6.7: Ejemplo de código escrito en CUDA C.

En los siguientes apartados se explica a detalle la aplicación del modelo de programación de CUDA C sobre cada uno de los algoritmos abordados en el capítulo 3.

6.3. Extracción de puntos característicos

En el Algoritmo 3.1 se detallan los pasos a seguir para calcular los vectores normales en una nube de puntos. Dicho algoritmo se ejecuta de manera secuencial, es decir, se ejecuta una instrucción detrás de otra, la Figura 4.3 ilustra este proceso. Dado el carácter secuencial de esta ejecución, el proceso tiene una duración de **n** (fila*columna) repeticiones. Esto implica que, un procesador deberá ejecutar **n** instrucciones (una por cada vector normal calculado) para terminar el proceso.

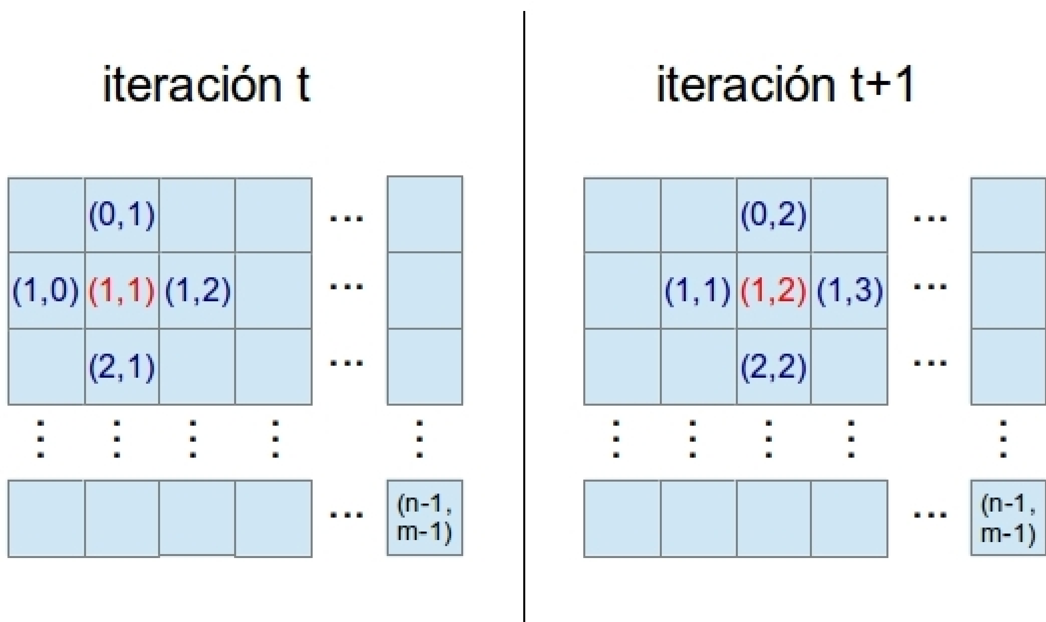

Figura 6.8: Diagrama representativo de los accesos a memoria ocurridos en cada iteración del cálculo de vectores normales. Las lecturas de memoria se representan en color azul, las escrituras en color rojo.

En cada iteración del proceso de cálculo de vectores normales de una escena 3D suceden dos cosas:

- Se realizan 4 lecturas en memoria (una por cada punto adyacente).
- Se realiza una escritura en memoria, correspondiente al vector normal calculado.

Cabe destacar que, en cada iteración, la escritura en memoria se realiza en una dirección de memoria diferente, Figura 6.9. Esto representa una gran ventaja al traducir el algoritmo secuencial a un algoritmo paralelo.

Para llevar a cabo la traducción del Algoritmo 3.1, se sustituyó el ciclo iterativo por una ejecución en paralelo, siguiendo los siguientes lineamientos:

- El cálculo de vectores normales se realizó, cada uno, en un núcleo del GPU. Se escribió un kernel para calcular un vector normal en un punto (este kernel será ejecutado por cada núcleo del GPU).
- Dado que cada escritura en memoria (una por cada vector normal calculado) se realiza en una localidad distinta a las demás, no existirán conflictos de escritura en memoria entre los distintos núcleos del GPU.
- Para calcular todos los vectores normales, el código del kernel debe ser ejecutado por n (fila*columna) núcleos.

El diagrama de flujo que representa al proceso de extracción de puntos característicos en paralelo se muestra en la Figura 6.9, donde:

- **P**: Corresponde a la estructura de datos que almacena a una nube de puntos. Mediante P(j) se denota el acceso al elemento j de la nube de puntos.
- **N**: Corresponde a la estructura de datos que almacena a una nube de vectores normales. Mediante N(j) se denota el acceso al elemento j de la nube de vectores normales.
- **k**: Es la cantidad total de núcleos que ejecutan el código escrito para el dispositivo.
- **Thread_Idx**: Es un identificador único asignado a cada núcleo del dispositivo. Mediante este identificador es posible conocer qué núcleo se encuentra ejecutando el código escrito para el dispositivo.
- **A, B**: Son vectores tangenciales a un punto P(j) de la nube de puntos P. Mediante estos vectores es posible calcular un vector normal al punto P(j).
- **getVec_A(punto)**, **getVec_B(punto)**: Son métodos que reciben como único parámetro un punto de la nube de puntos y calculan los vectores tangenciales A y B, respectivamente, a dicho punto (véase el apartado 3.2).
- **prodCruz(vectorA,vectorB)**: Es un método que calcula el vector normal a dos vectores tangenciales (véase el apartado 3.2). Recibe como parámetros dichos vectores tangenciales (vectorA y vectorB) y retorna el vector normal calculado.

El código de kernel es ejecutado por **n** procesadores en paralelo, en vez de ser ejecutado **n** veces por un mismo procesador de manera secuencial. La denominada *barrera de sincronización*, realiza una pausa la ejecución del código del equipo anfitrión hasta que todos los núcleos terminen la ejecución del kernel y hayan escrito sus resultados a memoria. Posteriormente se copian los resultados desde la memoria del dispositivo hacia la memoria del equipo anfitrión.

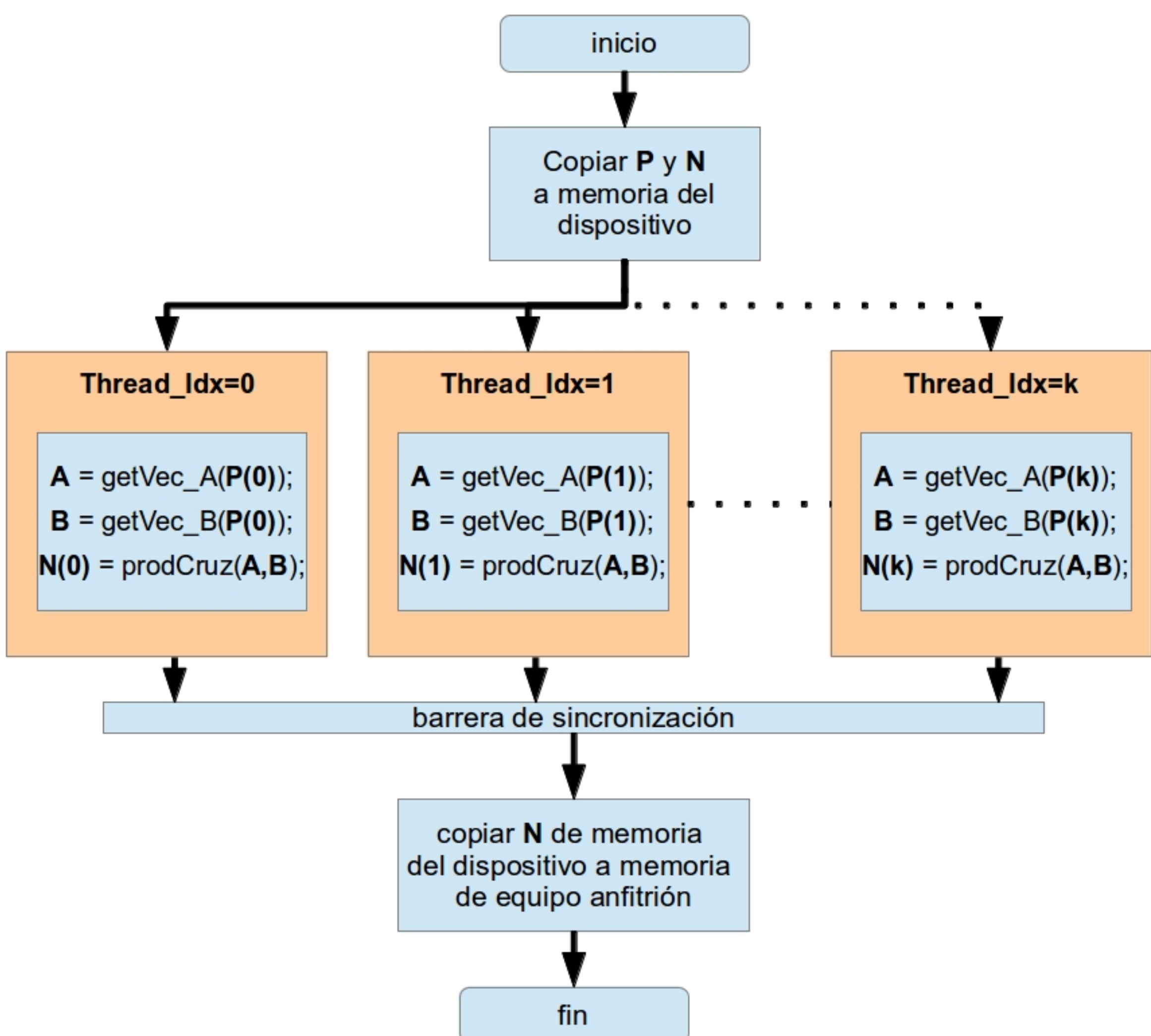

Figura 6.9: Diagrama de actividades que representa el flujo de ejecución (en paralelo) del proceso de extracción de características en una nube de puntos.

6.4. Segmentación de la superficie plana dominante

El algoritmo RANSAC, Algoritmo 3.2, es un proceso iterativo mediante el cual se obtiene un modelo matemático que describe a un determinado conjunto de datos. En este proceso es necesario realizar (en cada iteración) búsquedas a través de todo el conjunto de datos con el fin de hallar aquellos datos que empaten con el modelo matemático (cálculo de inliers). Realizar estas búsquedas puede llegar a ser una tarea computacionalmente costosa, en especial cuando se manejan grandes conjuntos de datos.

La Figura 4.5 muestra el diagrama de flujo correspondiente al cálculo de inliers. Al igual que el algoritmo de extracción de puntos característicos, se trata de un proceso secuencial en el que se requiere escribir y leer de memoria en cada iteración. Como se puede apreciar en el diagrama de la Figura 4.4, el cálculo de inliers se realiza en cada iteración del algoritmo RANSAC (mediante la llamada a la subrutina **obtener_inliers**).

Lo anterior se convierte en un cuello de botella cuando el conjunto de datos es muy grande o cuando se realizan un gran número de iteraciones del algoritmo RANSAC. Una nube de puntos (con una resolución de 640*480) puede contener hasta 307200 datos; realizar múltiples búsquedas sobre un conjunto de datos de ese tamaño resulta computacionalmente costoso.

Para acelerar la ejecución de este proceso, se llevó a cabo la traducción de la ejecución secuencial de la subrutina **obtener_inliers** a una ejecución en paralelo. La Figura 6.10 muestra el diagrama de flujo de la ejecución en paralelo implementada para la subrutina **obtener_inliers**, donde:

- **P**: Corresponde a la estructura de datos que almacena a una nube de puntos. Mediante P(j) se denota el acceso al elemento j de la nube de puntos.
- **M**: Variable correspondiente a la estructura de datos que contiene los valores de los parámetros necesarios para calcular una superficie plana en un sistema coordenado de tres dimensiones. En esta variable se encuentran almacenados los parámetros obtenidos en una iteración del algoritmo RANSAC.
- **D**: Corresponde al valor máximo de distancia euclídea que un punto de la nube de puntos puede alcanzar respecto al plano **M**. Los puntos más allá de esta distancia no son considerados como puntos pertenecientes al plano (outliers), mientras que los puntos por debajo de esta distancia sí los son (inliers).
- **IC**: Es un vector de datos el el cual se almacena un 1 por cada punto inlier (y que además de encuentre orientado en la misma dirección que el plano **M**) de la nube de puntos y un 0 por cada outlier.
- **NK**: Es la cantidad total de núcleos que ejecutan el código escrito para el dispositivo.

- **dist_euclidea(punto, modelo)**: Es un procedimiento almacenado que calcula la distancia euclídea desde un punto 3D a un plano 3D. Recibe dos parámetros: las coordenadas cartesianas del punto y el modelo matemático del plano.
- **orientacion(punto, modelo)**: Es un procedimiento almacenado que recibe como parámetros un punto 3D y el modelo matemático de una superficie plana. Retorna **true** si el punto y la superficie descrita por el modelo tienen la misma orientación y **false** en otro caso.

A continuación se detalla el flujo de procesamiento que sigue el diagrama de la Figura 6.10:

1. Las variables **P**, **M** y **D** (correspondientes a la nube de puntos, la ecuación del plano y al umbral de distancia máxima) son obtenidas desde el proceso principal del algoritmo RANSAC (Figura 4.4).
2. Se realiza una copia de las variables **P**, **M** y **D** a la memoria del dispositivo. Los datos que almacenan estas variables serán requeridas, por cada uno de los núcleos del GPU, más adelante en la ejecución.
3. Se crea el vector contador de inliers (variable **IC**, por Inliers Counter) y se almacena en memoria compartida (debido a que distintos núcleos del GPU necesitarán leer y almacenar datos en este vector).
4. Se ejecuta el código del kernel en cada núcleo de la GPU. El kernel se encuentra compuesto por las siguientes etapas de ejecución:
    1. Se calcula la distancia euclídea (mediante la subrutina **dist_euclidea**) desde cada punto de la nube de puntos hacia el plano descrito por la ecuación del plano **M**.
    2. Se verifica si la orientación del punto analizado y el plano son la misma. Esto se realiza mediante la subrutina **orientacion**.
    3. Si el punto se encuentra cercano al plano (a una distancia menor que el umbral D) y ambos tienen la misma orientación, se escribe en el vector **IC** un 1, en caso contrario se escribe un 0.
    4. La primera barrera de sincronización pausa la ejecución de los núcleos de la GPU, la ejecución se reanuda hasta que todos los demás núcleos alcancen este punto. Lo anterior permite que, al pasar la barrera, todos los núcleos hayan escrito su resultado en el vector **IC**.
    5. Se realiza una suma por reducción de todos los datos del vector **IC** (que ahora contiene un 1 por cada inlier y un 0 por cada outlier).
5. Se copia, a la memoria del equipo anfitrión, el resultado de la suma del vector **IC**. Este resultado representa la cantidad de inliers que existen en la nube de puntos **P** para el modelo **M**.

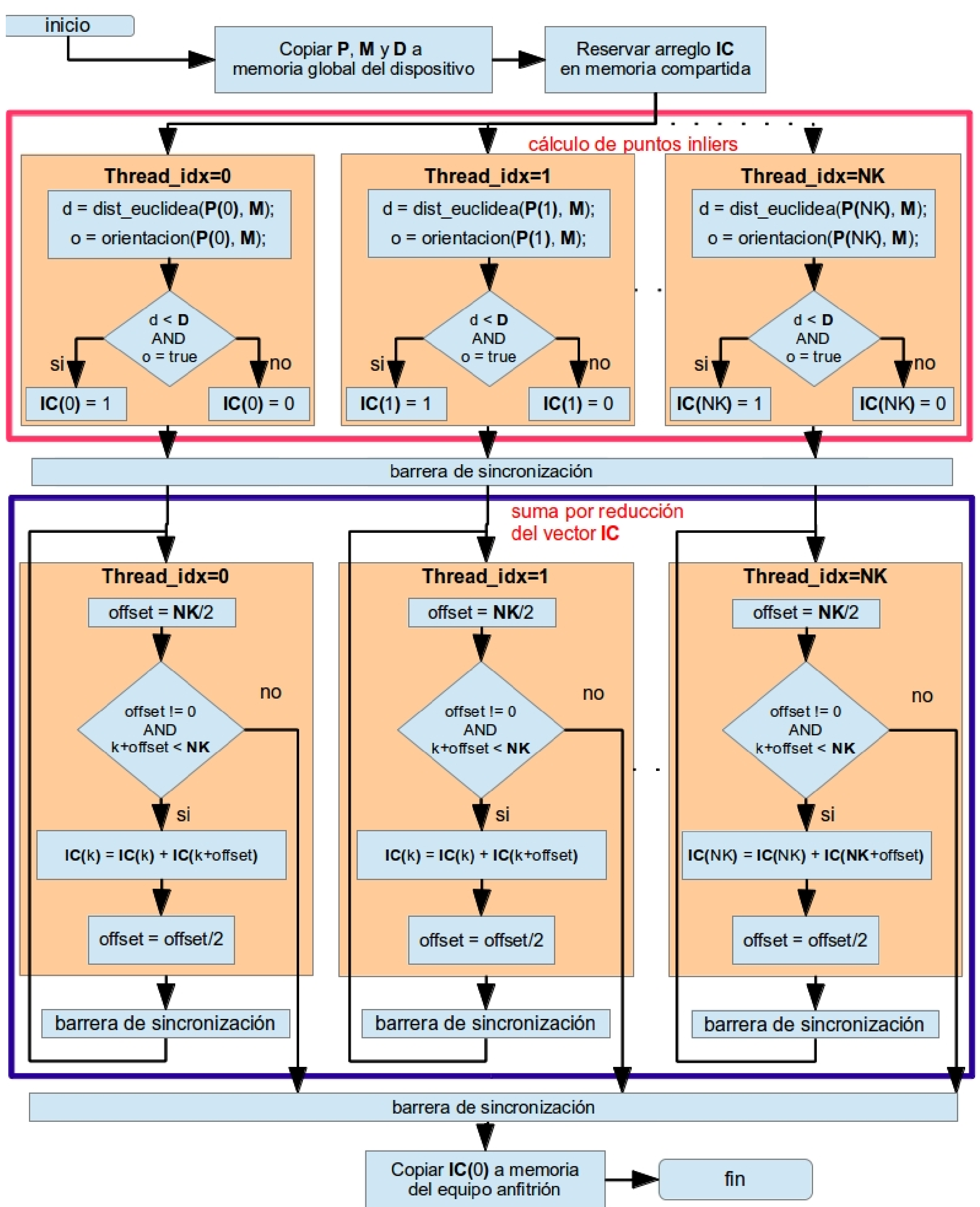

Figura 6.10: Diagrama de actividades que representa el flujo de ejecución (en paralelo) del proceso de cálculo de inliers.

Capítulo 7. Experimentos y resultados

En este capítulo se presentan una serie de comparaciones entre las implementaciones de los algoritmos, descritos en el Capítulo 3, en las tres diferentes arquitecturas (CPU, GPU y FPGA) abordadas en este trabajo. Para llevar a cabo dichas comparaciones, se hizo uso de un CPU Intel Core i5, un CPU Intel Atom N2600, un FPGA Altera Cyclone IV y un GPU Nvidia GeForce 9600GT. Las escenas 3D procesadas pertenecen a un conjunto de datos de prueba creado por investigadores de la Universidad de Bonn, véase [OSWSB11]. Dicho conjunto de escenas fue creado realizando tomas, en interiores, utilizando una cámara RGB-D Microsoft Kinect; cada escena 3D cuenta con una resolución de 640x480 píxeles. Las comparaciones realizadas consisten en: tiempo de ejecución, perdida de precisión de datos y precisión del proceso.

7.1. Comparación del tiempo de ejecución

En robótica móvil, uno de los aspectos más importantes es el tiempo de ejecución de los procesos dado que, en ocasiones, se requiere que un robot interactúe con humanos en ambientes dinámicos, garantizando la integridad física de los usuarios. Es por esto que, aplicaciones con tiempos de respuesta inmediatos permiten que un robot pueda actuar de forma instantánea ante repentinos cambios en el ambiente para así, evitar accidentes (colisionar con el operador o colisionar con objetos) y ofrecer comportamientos socialmente aceptables [GFS07]. Debido a lo anterior, es importante, llevar a cabo mediciones en cuanto al tiempo de ejecución de las aplicaciones diseñadas para procesos de robótica móvil.

7.1.1. Extracción de puntos característicos

En este apartado se presenta la comparativa, del tiempo de ejecución, de las implementaciones correspondientes al Algoritmo 3.1 (*Cálculo de vectores normales en una nube de puntos organizada.*) en cada uno de los sistemas (CPU multi-core, CPU+FPGA y CPU+GPU) abordadas en este trabajo. La Figura 7.3 muestra el promedio de las mediciones realizadas con distintas resoluciones de la escena 3D observada: VGA (640*480 píxeles), QVGA (640*480 píxeles) y QQVGA (640*480 píxeles). La Tabla 7.1 muestra los valores numéricos de estas mediciones.

Para el sistema CPU+FPGA se realizaron pruebas con la arquitectura mostrada en el apartado 5.2 con dos implementaciones distintas: con segmentación de cause (pipe-line) y sin segmentación de cauce (no pipe-line).

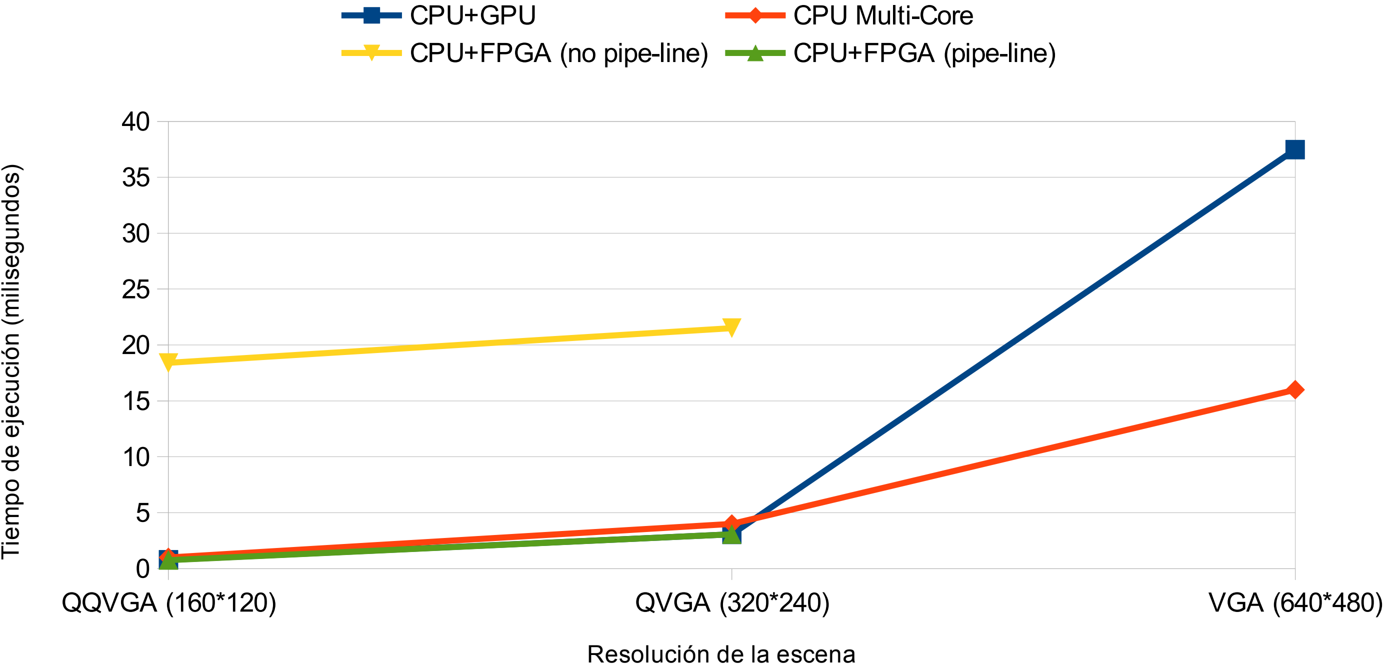

Figura 7.1: Comparativa del tiempo de ejecución para el proceso de extracción de puntos característicos.

	CPU+GPU	CPU Multicore	CPU+FPGA (pipe-line)	CPU+FPGA (no pipe-line)
QQVGA (160*120)	0.79 ms	1 ms	0.76 ms	18.4 ms
QVGA (320*240)	3.06 ms	4 ms	3.076 ms	21.5042 ms
VGA (640*480)	37.47 ms	16 ms		

Tabla 7.1: Comparativa del tiempo de ejecución para el proceso de extracción de puntos característicos.

7.1.2. Segmentación de la superficie plana dominante

En este apartado se presenta la comparativa, del tiempo de ejecución, de las implementaciones correspondientes al Algoritmo 3.2 (*RANSAC aplicado a una escena 3D.*) en los sistemas CPU multi-core y CPU+GPU. La Figura 7.2 muestra el promedio de las mediciones realizadas con distintas resoluciones de la escena 3D observada: VGA (640*480 píxeles), QVGA (640*480 píxeles) y QQVGA (640*480 píxeles). La Tabla 7.2 muestra los valores numéricos de estas mediciones.

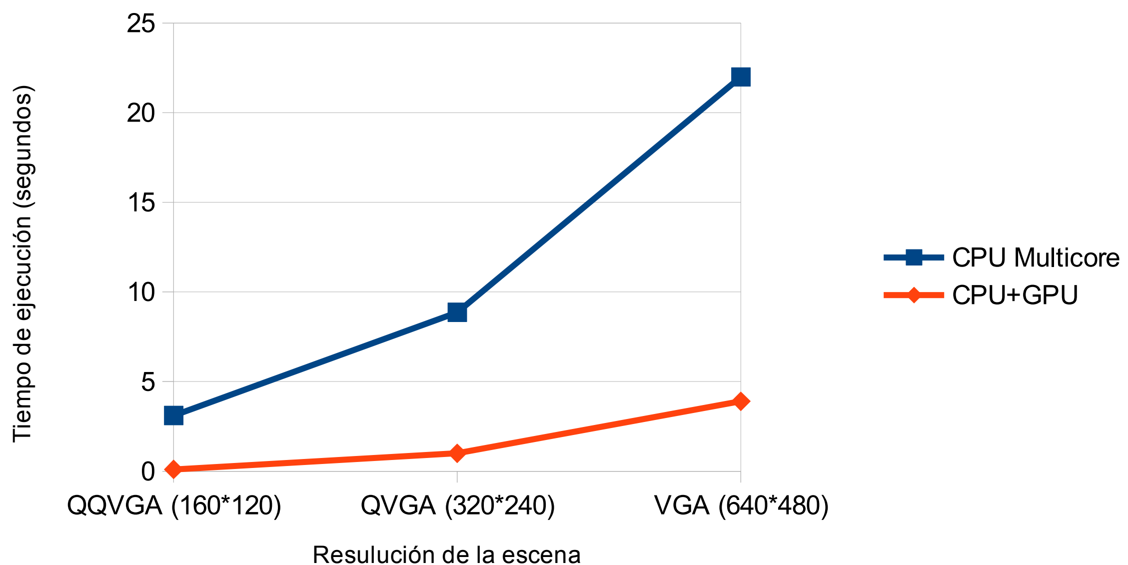

Figura 7.2: Comparativa del tiempo de ejecución para el proceso de segmentación de la superficie plana dominante.

	CPU Multicore	**CPU+GPU**
QQVGA (160*120)	3.1 s	0.097 s
QVGA (320*240)	8.86 s	1.27 s
VGA (640*480)	21.99 s	3.9 s

Tabla 7.2: Comparativa del tiempo de ejecución para el proceso de segmentación de la superficie plana dominante.

7.2. Comparación de la precisión de datos

En este trabajo, la precisión de datos se refiere a la capacidad de almacenamiento, en bits, de un determinado tipo de dato. Cuando la ejecución de un proceso se lleva a cabo en una sola arquitectura de hardware, es posible mantener la precisión de datos intacta (por ejemplo, al hacer uso de un único tipo de dato para todas las variables). Sin embargo, al trabajar con múltiples arquitecturas, es probable que los datos experimenten ciertos cambios de precisión causados por incompatibilidades entre las distintas arquitecturas involucradas. En este apartado, se muestran las mediciones llevadas a cabo sobre los datos resultantes de la ejecución del algoritmo 3.1 para determinar la cantidad de error acumulado en los datos debido a la perdida de precisión.

7.2.1. Extracción de puntos característicos

Para medir la pérdida de precisión en los datos resultantes del proceso de extracción de puntos característicos se realizó el cálculo del error cuadrático medio (ECM) entre los puntos característicos extraídos por el CPU Multicore (como punto de comparación) y los puntos característicos extraídos por el resto de los sistemas. Los resultados de estas mediciones se muestran en la Tabla 7.3.

Se eligieron los puntos característicos extraídos por el CPU Multicore como punto de comparación por dos motivos:

1. Los datos de muestra fueron adquiridos mediante esta arquitectura.
2. En esta arquitectura, se usó el mismo tipo de dato para la adquisición de las escenas y para la extracción de los puntos característicos: coma flotante de 64 bits de longitud.

	VGA (640*480)	**QVGA (320*240)**	**QQVGA (160*120)**
CPU Multicore	0	0	0
CPU+GPU	3.76E-083	4.56E-086	1.05E-081
CPU+FPGA		4.57E-086	3.62E-080

Tabla 7.3: Cálculo del ECM entre los distintos sistemas.

7.3. Comparación de la precisión del proceso

Al procesar un conjunto de datos, sin aparente significado alguno, es posible extraer cierta información de ellos. Sin embargo puede ocurrir, en ocasiones, que la información extraída no siempre sea la correcta. Esto suele suceder por distintos motivos: cuando se cuenta con información redundante en el conjunto de datos, debido al ruido existente en los datos, debido a la perdida de resolución, etc. En procesamiento de nubes de puntos no se está exento de las situaciones descritas anteriormente, por tal motivo, es importante analizar los datos resultantes de la ejecución de los algoritmos 3.1 y 3.2 sobre las nubes de puntos y determinar la certeza del resultado obtenido (precisión del proceso).

7.3.1. Extracción de puntos característicos

Los vectores normales obtenidos de una nube de puntos nos dan información acerca de la orientación espacial de cada punto perteneciente a la nube. Cada vector normal obtenido, dado que es un vector 3D, está compuesto por tres componentes (x,y,z). Para representar gráficamente a un conjunto de vectores normales correspondiente a una escena 3D se suele utilizar una representación como la que se muestra en la Figura 7.3, en la que cada línea de color rojo corresponde a un vector normal.

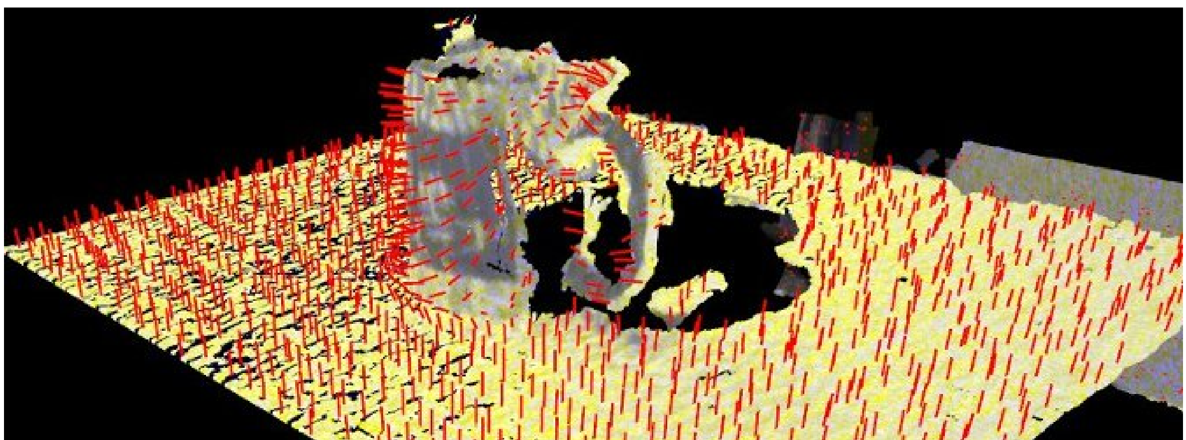

Figura 7.3: Representación gráfica de los vectores normales de una escena 3D.

Este tipo de representación, permite observar gráficamente la orientación de cada punto en la escena 3D, sin embargo, no permite comparar los puntos característicos pertenecientes a dos escenas distintas (debido a que esta representación es gráfica únicamente). Otro tipo de representación consiste en asignar un valor de color (r,g,b) a cada componente (x,y,z) de cada vector normal, Figura 7.4. Mediante este tipo de representación es posible observar gráficamente aquellos vectores normales con orientación similar (color similar) y realizar comparaciones con otros conjuntos de datos (mediante las componentes r,g,b).

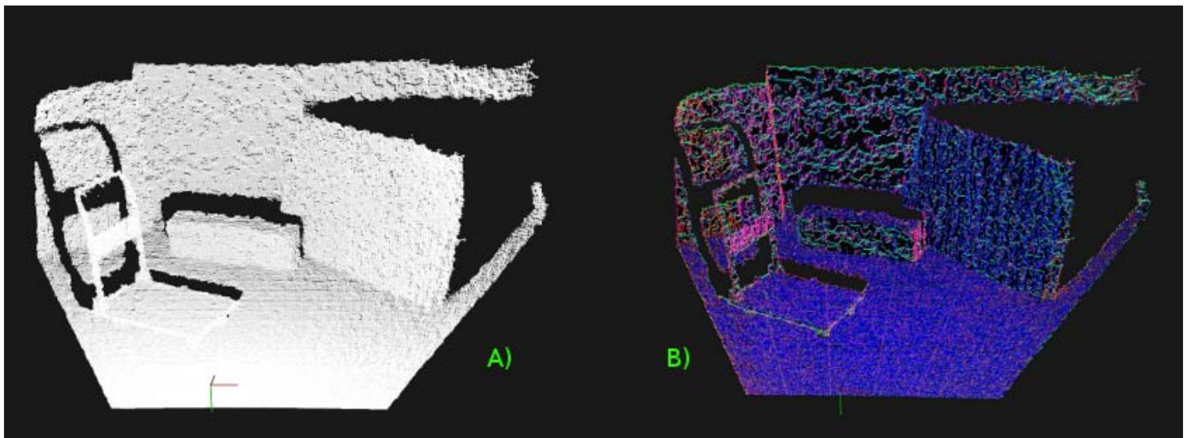

Figura 7.4: Representación RGB de puntos característicos.

Para llevar a cabo las mediciones de precisión entre los vectores característicos extraídos por los distintos sistemas se hizo uso de la métrica conocida como Similitud Estructural (SSIM, véase [WBSS04]). Esta métrica consiste en determinar la calidad de una imagen basándose en una imagen inicial no comprimida y libre de distorsión. Para esto, se parte de la premisa de que los píxeles espacialmente cercanos cuentan con fuertes interdependencias estructurales. Debido a lo anterior, SSIM se aplica en distintas ventanas de una imagen (cada ventana conteniendo a un conjunto de píxeles espacialmente cercanos). La siguiente fórmula es aplicada entre dos ventanas x y y de las imágenes a comparar:

$$SSIM(x, y) = \frac{(2\mu_x \mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)}$$

donde:

- μ_x es el promedio de la ventana x.
- μ_y es el promedio de la ventana y.
- σ_x^2 es la varianza de la ventana x.
- σ_y^2 es la varianza de la ventana y.
- σ_{xy} es la covarianza entre x y y .
- y con $c_1 = (k_1 L)^2, c_2 = (k_2 L)^2$; donde L es el rango dinámico[2] del valor de los píxeles; y donde $k_1 = 0.01$ y $k_2 = 0.03$ por definición.

2 Se refiere al rango de diferencia entre los valores mayor y menor que cierta cantidad pueda adquirir. Para el caso de un píxel con formato de escala de grises este rango es: 255-0.

El resultado de aplicar SSIM sobre dos imágenes es un valor decimal entre 0 y 1. Donde 1 es alcanzable en el caso en que ambas imágenes son idénticas. Comúnmente se hace uso de ventanas de tamaño 8x8 para calcular SSIM, idealmente se utiliza una ventana por píxel.

La métrica descrita anteriormente es utilizada para hallar el factor de similitud entre las distintas imágenes que representan los puntos característicos extraídos de una escena (mediante la ejecución del algoritmo 3.1). Se toma como imagen inicial la imagen RGB obtenida por el CPU Multicore (debido a que esta no presenta pérdida de precisión de datos) y se aplica SSIM sobre cada componente de color entre la imagen original y las imágenes obtenidas por los sistemas CPU+FPGA y CPU+GPU. La Tabla 7.4 muestra los resultados de estas mediciones; como se mencionó en el párrafo anterior, el valor de similitud se encuentra en el rango $[0,1]$ pudiendo llegar hasta 1 cuando ambas imágenes son idénticas.

	VGA (640*480)	**QVGA (320*240)**	**QQVGA (160*120)**
CPU Multicore	1	1	1
CPU+GPU	0.9999401	0.99970906	0.9874038
CPU+FPGA		0.8794	0.859

Tabla 7.4: Cálculo de SSIM sobre las imágenes RGB generadas por los diferentes sistemas.

Capítulo 8. Conclusiones y trabajo futuro

Llevar a cabo la implementación de algoritmos en sistemas de cómputo heterogéneo implica resolver distintos problemas como la compatibilidad entre datos, sincronización de procesos y la transferencia de datos entre las múltiples arquitecturas involucradas. El nivel de complejidad de estos problemas puede variar dependiendo del tipo de sistema en el cual se presentan y de las arquitecturas involucradas.

En un sistema de cómputo heterogéneo compuesto por múltiples CPU (pertenecientes al mismo fabricante y además encapsulados en un mismo chip) los problemas descritos anteriormente no representan obstáculo alguno. Lo anterior debido a que el fabricante del chip los resuelve a nivel de hardware y el sistema operativo se encarga de manejarlos a nivel de software, haciendo todo transparente para el desarrollador. En un sistema CPU-GPU ocurre algo similar, debido a que existen estándares de hardware que aseguran la compatibilidad entre arquitecturas (CPU y GPU) y software (como CUDA) que permite realizar transferencia de datos y sincronización entre procesos de manera sencilla.

Por otro lado, para dar solución a estos problemas en un sistema CPU-FPGA, es necesario diseñar una interfaz de hardware (en el FPGA) que resuelva las incompatibilidades entre datos, permita la sincronización entre procesos y permita la transferencia de datos. Asimismo, se deben diseñar módulos de software (en el CPU) que interactúen con el hardware descrito en el FPGA.

Las aplicaciones desarrolladas para funcionar en sistemas de cómputo heterogéneo requieren, la mayoría del tiempo, transferir datos entre ellas. Cuando las arquitecturas involucradas manejan distintos tipos de datos, puede ocurrir pérdida de información durante la transferencia y durante el procesamiento de los datos en cada una de las arquitecturas.

En un sistema CPU Multicore, no existe esta pérdida de información por transferencia de datos. Esto debido a que cada *core* del CPU maneja los mismos tipos de datos y, además, estos datos no son transferidos de una arquitectura a otra sino que se almacenan en una memoria global accesible por todos los *cores*. En el sistema CPU-GPU existe una pérdida mínima de información (ocasionada por la incompatibilidad en manejo de los tipos de datos), sin embargo, esta pérdida de información resulta ser insignificante debido a que tanto el CPU como la GPU trabajan con tipos de datos similares. Asimismo, en el sistema CPU-FPGA la pérdida de información resultó ser mínima (a pesar de pasar de una representación de 64 bits a una representación de 15 bits) esto debido a que los datos procesados no suelen ocupar 64 bits y pueden ser almacenados en tipos de datos de 15 bits sin experimentar una pérdida significativa de información.

Sin embargo, a pesar de existir una casi insignificante pérdida de información en los sistemas CPU-GPU y CPU-FPGA, está pequeña pérdida de información a nivel de datos causa un gran impacto al momento de representar gráficamente los resultados. Esto puede observarse en la Tabla 7.4, en la cual se mide la similitud (a nivel gráfico) entre los resultados obtenidos por los distintos sistemas. En esta tabla se puede observar que, a medida que la resolución de las imágenes disminuye, también disminuye la similitud entre los datos originales y los datos resultantes del procesamiento entregados por los distintos sistemas. Se puede apreciar también que, los datos resultantes del procesamiento realizado en el sistema CPU-FPGA fueron los que experimentaron una mayor disimilitud con los datos originales, la cuál se reflejó más notoriamente en la representación gráfica que en la representación numérica.

En cuanto al tiempo de ejecución, para el proceso de segmentación de la superficie plana dominante, el sistema CPU-GPU presentó el mejor rendimiento en todas las resoluciones de escena. Para el problema de extracción de puntos característicos los distintos sistemas de cómputo heterogéneo utilizados en este trabajo mostraron un desempeño similar.

Para las escenas con resolución QQVGA y QVGA el sistema CPU-GPU reportó el mejor desempeño, seguido por el sistema CPU-FPGA (pipe-line), el sistema de CPU Multicore y por último el sistema CPU-FPGA (no pipe-line). El sistema CPU-GPU presenta el mejor desempeño debido a su alto nivel de paralelismo, lo cual le permite procesar grandes cantidades de datos en poco tiempo. El sistema CPU-FPGA (pipe-line) se desempeña de manera eficiente dado que la arquitectura pipe-line implementada en el FPGA presenta cierto nivel paralelismo, además de ser una arquitectura dedicada exclusivamente a la extracción de puntos característicos. El sistema CPU-FPGA (no pipe-line) presenta un desempeño muy pobre debido a que, a pesar de contener una arquitectura dedicada en el FPGA, no tiene nivel de paralelismo alguno.

En cuanto a la resolución VGA no fue posible probar el desempeño en el sistema CPU-FPGA debido a que el tamaño de la escena excedía la capacidad de almacenamiento del FPGA. En cuánto a los sistemas CPU-GPU y CPU Multicore, el CPU Multicore presentó menor tiempo de ejecución. Lo anterior debido a que la cantidad de datos (307200) excedía por mucho el número de núcleos de la GPU (512), lo cuál provocó que se desaprovechara tiempo de cómputo en el GPU reasignando los núcleos, mientras que el CPU se dedicó únicamente a la ejecución del proceso.

Mediante las mediciones realizadas, se demostró que un sistema con una arquitectura de hardware específica implementada en un FPGA, es capaz de competir en rendimiento con sistemas como las GPU y los CPU del mercado. Sin embargo, debido a las limitaciones de los FPGA (poca capacidad de almacenamiento y poca velocidad de reloj) no se pudo mostrar al 100% lo eficiente que una arquitectura de hardware dedicada puede llegar a ser. Una implementación física (un circuito ensamblado con componentes electrónicos) puede llegar a superar estas limitantes y obtener un mejor rendimiento.

Los tres sistemas de cómputo heterogéneo probados en este trabajo demostraron tener un buen desempeño en el procesamiento de nubes de puntos. Como trabajo futuro, se propone implementar un sistema de cómputo heterogéneo combinando las tres arquitecturas usadas en este trabajo (CPU-FPGA-GPU) de la siguiente manera:

- Para la adquisición de datos sería ideal hacer uso de un sensor RGB-D conectado a un FPGA el cual, debido a su alta flexibilidad, permitiría ejecutar distintos algoritmos de pre-procesamiento (como algoritmos de sub-muestreo) en los datos adquiridos con tiempos de respuesta relativamente cortos.
- Posteriormente, realizar la transferencia de datos pre-procesados a un CPU (eliminando así la necesidad de almacenamiento en el FPGA).
- Obtenidos los datos pre-procesados del FPGA, un sistema CPU-GPU podría ejecutar algoritmos de inteligencia artificial (por parte del CPU) y algoritmos que requieran gran poder de cómputo (por parte del GPU).

Apéndice A. Sistema de transmisión y recepción de datos entre CPU y FPGA

En este anexo se detalla el sistema de hardware-software desarrollado, bajo el sistema operativo Ubuntu 12.04, para la transmisión de datos entre el CPU y el FPGA, a través del puerto PCI Express de la tarjeta Intel DE2i-150. Este sistema fue desarrollado haciendo uso de los lenguaje de programación C++ y VHDL.

A.1 Tarjeta de desarrollo Intel De2i-150

La tarjeta Intel DE2i-150 es una plataforma integrada que combina un procesador Intel Atom N2600 con un FPGA Altera Cyclone IV GX. La DE2i-150 es un sistema de computadora completa que reúne al mundo del procesamiento de alto rendimiento con el de hardware reconfigurable. En la Figura A.1 se muestra el diagrama de bloques en alto nivel que representa la arquitectura de la tarjeta de desarrollo DE2i-150. El procesador Intel Atom y el FPGA Altera se encuentran conectados mediante un puerto PCI Express, cada uno de estos componentes cuentan con diferentes dispositivos periféricos.

El CPU cuenta con los siguientes dispositivos periféricos:

- Una interfaz Ethernet para la transmisión de datos.
- Entradas y salidas de audio.
- Salida de video mediante VGA y HDMI.
- Puertos USB 2.0.
- Conectores SATA.

Por otra parte, el FPGA cuenta con:

- Entradas de video RCA.
- Salida de video mediante VGA.
- Una interfaz Ethernet para la transmisión de datos.
- Pines de Entrada/Salida de Propósito General (GPIO, *General Purpose Input Output*).
- Una pantalla led de 7 segmentos.
- Un receptor infrarrojo.
- Un puerto RS232.
- Puertos USB 2.0.

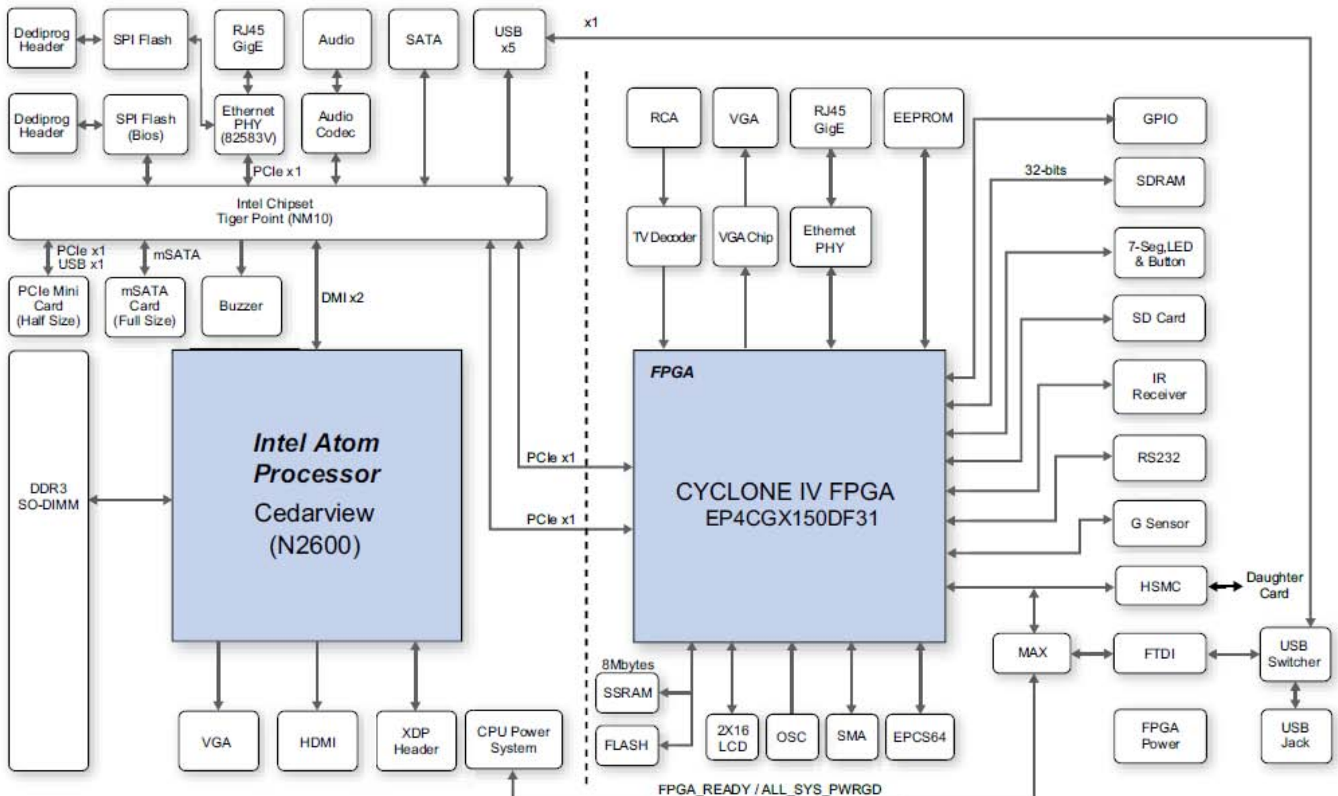

Figura A.1: Arquitectura de alto nivel de la tarjeta de desarrollo Intel DE2i-150.

A.1.1. El procesador Intel Atom N2600

El Intel Atom N2600 es un procesador de doble núcleo comúnmente usado en dispositivos portátiles como netbooks y tabletas electrónicas. Cuenta con una arquitectura de 64 bits, trabaja con una frecuencia de reloj de 1600 MHz, cuenta con una tarjeta gráfica integrada PowerVR SGX545 (400 MHz) y soporta hasta un máximo de 2GB de memoria RAM.

Las especificaciones completas de este procesador pueden ser consultadas a detalle en la página web: http://ark.intel.com/es/products/58916

A.1.2. El FPGA Altera Cyclone IV EP4CGX150DF31

El FPGA Cyclone IV es un dispositivo reconfigurable desarrollado por la compañia Altera. Este dispositivo cuenta con un total de 149,760 elementos lógicos programables, 6,480 bits de memoria, 360 multiplicadores de 18 bits, 8 transcivers, 8 PLL, 475 pines de entrada/salida para el usuario y trabaja a una frecuencia de reloj de 50MHz. Los detalles de su funcionamiento, así como otras especificaciones, pueden ser consultados en la página web: http://www.altera.com/literature/manual/rm_civgx_fpga_dev_board.pdf

A.2 Sistema de hardware-software para transferencia de datos

A pesar de que el FPGA y el CPU se encuentran conectados mediante el puerto PCI Express de la tarjeta DE2i-150, estos no se reconocen uno al otro y por lo tanto no se puede llevar a cabo transferencia de datos entre estos dos dispositivos, para lograrlo es necesario notificar al CPU que el FPGA existe y vice-versa. Esto implica desarrollar una serie de pasos:

1. Desarrollar un controlador de software para el CPU, este controlador permitirá al CPU identificar que el FPGA se encuentra conectado.
2. Desarrollar una interfaz de hardware en el FPGA, esto le permitirá al FPGA administrar los datos transferidos mediante el bus PCI Express.
3. Desarrollar un sistema de sincronización entre el hardware del FPGA y el software del CPU, para sincronizar el envío y recepción de datos entre el CPU y el FPGA.

A.2.1. Controlador de software

Se adaptó el controlador de software desarrollado por Patrick Schaumont en 2013 (sitio web del proyecto: http://rijndael.ece.vt.edu/de2i150/designs/hellopci.pdf) para su funcionamiento en el sistema operativo Ubuntu 12.04. Este controlador permite el envío de secuencias de 32 bits de datos y recepción de 15 bits de datos por medio del puerto PCI Express de un procesador.

A.2.2. Interfaz de hardware

Para el desarrollo de la interfaz de hardware en el FPGA se hizo uso del sistema de desarrollo Qsys de Altera. Qsys permite la construcción de módulos de hardware a partir de diagramas de bloque, los diseños realizados en Qsys son compilados y convertidos a código VHDL o Verilog. En la Figura A.2 se muestra el sistema diseñado para la adquisición y envío de datos desde el FPGA al CPU, con este hardware el FPGA puede transmitir secuencias de 15 bits y recibir secuencias de 32 bits de longitud.

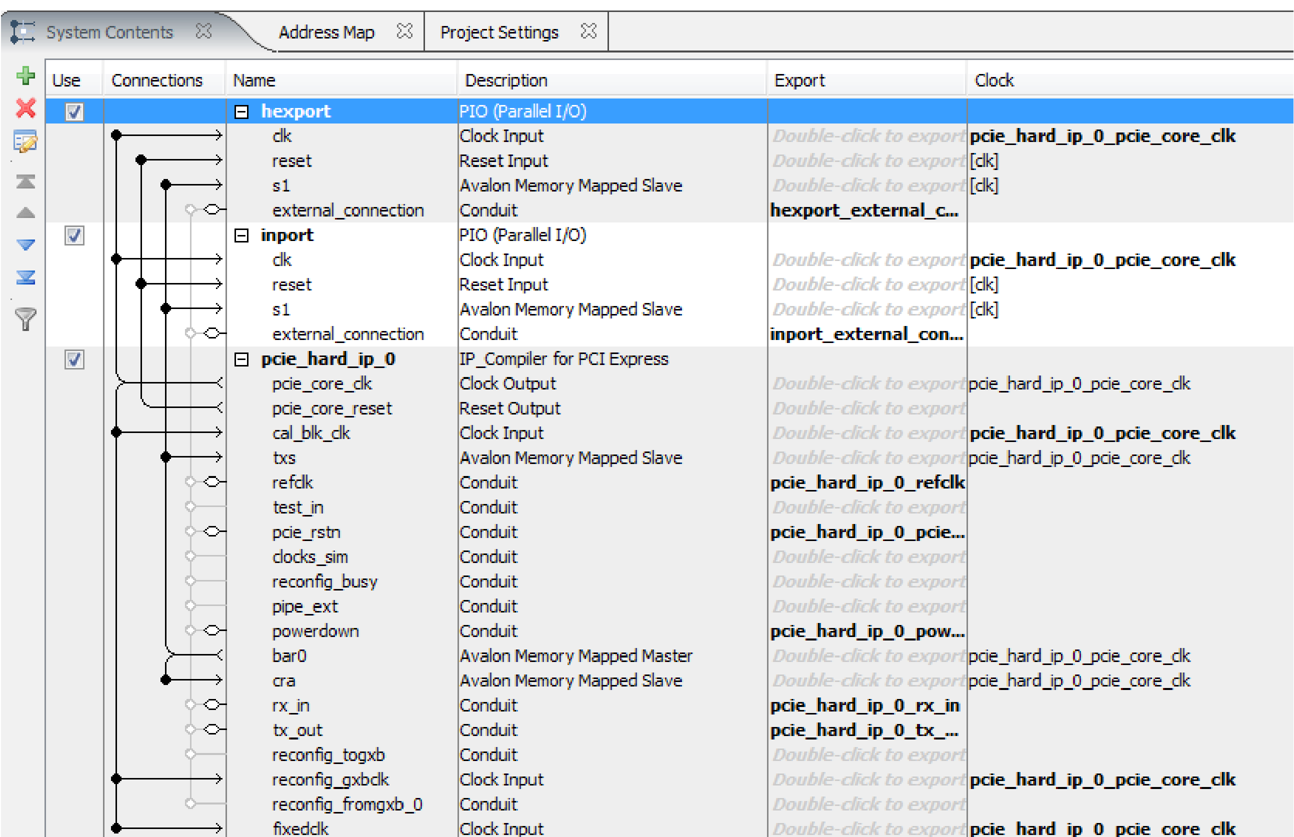

Figura A.2: Sistema de Hardware desarrollado para la adquisición y envío de datos desde el FPGA hacia el CPU.

A.2.3. Sistema de sincronización hardware-software

La comunicación entre dos arquitecturas diferentes (FPGA y CPU) se debe realizar de manera ordenada (dado que sólo se tiene un canal de comunicación) es decir, mientras una arquitectura transmite datos la otra únicamente debe recibirlos. También existe el problema de que tanto el CPU como el FPGA trabajan a diferentes frecuencias de reloj (CPU a 1600MHz y FPGA a 50 MHz).

Para sincronizar el envío y recepción de datos entre estas dos arquitecturas se desarrolló un sistema de Hardware-Software basado en la arquitectura Cliente-Servidor. En este sistema, el CPU actúa como Cliente (dado que es el que realiza la adquisición de datos y por lo tanto hace peticiones de procesamiento al FPGA) y el FPGA actúa como Servidor (atiende las peticiones de procesamiento demandadas por el CPU). El sistema de comunicación se encuentra especificado en alto nivel por el diagrama de la Figura A.3.

Cada paquete de datos enviados desde el CPU hacia el FPGA tiene una longitud de 32 bits, de los cuales los 4 bits más significativos se encuentran reservados para indicar el estado de la transmisión. El formato de la trama de datos es el siguiente:

31 0
[0 | 1 0 1 | 0 0 ……. 0]

El bit 31 indica la transmisión de un nuevo dato, es decir, con cada nueva trama enviada este bit modifica su actual valor por su complemento. Los bits [30,29,28] indican el estado de la transmisión, la Tabla A.1 muestra los estados de la transmisión y su representación binaria. Finalmente, los bits [27 … 0] son usados únicamente para datos.

Los paquetes de datos enviados desde el FPGA al CPU tienen una longitud de 16 bits y únicamente contienen datos. Antes de comenzar el envío de datos desde el FPGA al CPU el CPU debe recibir un indicador, dicho indicador se conforma por la cadena de bits:

15 0
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]

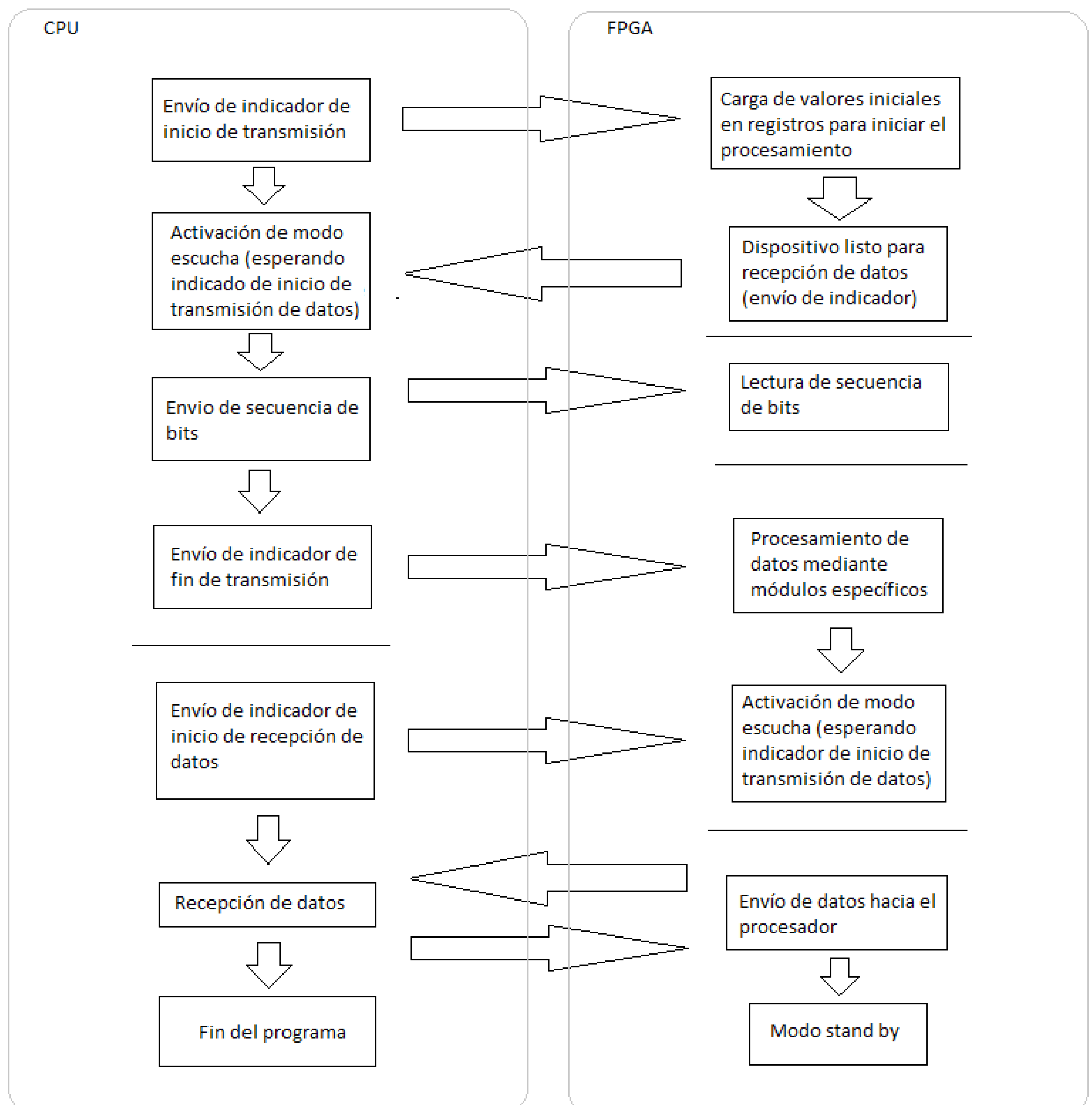

Figura A.3: Diagrama de alto nivel que representa el proceso de comunicación entre el CPU y el FPGA.

Bits [30,29,28]	Estado
000	No hay transmisión (el CPU se encuentra ejecutando algún proceso distinto al envío de datos).
001	Estado que indica el inicio de la transmisión (el CPU comenzará a transmitir datos).
010	Estado que indica que la trama actual contiene un dato de 28 bits.
011	Estado que indica el fin de la transmisión (el CPU no enviará mas datos y, por lo tanto, el FPGA comenzará con el procesamiento de datos).
101	Estado que indica que el CPU está listo para leer un dato desde el FPGA (si el FPGA no ha terminado la etapa de procesamiento, el CPU quedará en espera).
110	Estado que indica que el CPU ha leído un dato. Es importante que el CPU notifique al FPGA cada vez que un dato ha sido leído para que el FPGA prepare el nuevo dato a enviar.
111	Estado que indica que el CPU ha terminado de leer datos (el CPU le indica al FPGA que no leerá mas datos).

Tabla A.1: Tabla de estados para la transmisión de datos desde el CPU hacia el FPGA

Índice de figuras

Figura 1.1: Robot de servicio Justina, proyecto desarrollado en el laboratorio de BioRobótica de la UNAM..14

Figura 2.1: Ejemplo de nube de puntos.................15

Figura 2.2: Componentes principales y estructura interna de un CPU...16

Figura 2.3: Evolución de la Frecuencia de Reloj de los procesadores desde 1975.................................17

Figura 2.4: Computadora de von Neuman.............18

Figura 2.5: Arquitectura SIMD.............................19

Figura 2.6: Arquitectura MIMD............................20

Figura 2.7: Diagrama de alto nivel que representa la estructura interna de un FPGA..............................21

Figura 2.8: Diagrama de bloques de un CLB........22

Figura 2.9: Diagrama del flujo de diseño de un FPGA...23

Figura 2.10: Comparativa entre las arquitectura interna de un CPU y una GPU..............................24

Figura 2.11: Diagrama de bloques de un Streaming Multiprocessor..25

Figura 2.12: Diagrama de alto nivel de una GPU..26

Figura 2.13: Tarjeta de desarrollo Intel DE2i-150. Cuenta con un procesador Intel Atom y un FPGA Altera Cyclone IV...27

Figura 2.14: Patrón infrarrojo proyectado por una cámara RGB-D...29

Figura 2.15: Representación de la organización de una nube de puntos..30

Figura 3.1: Robot móvil de servicio Justina intentando detectar objetos situados sobre una superficie plana...32

Figura 3.2: Vecindad Pi (puntos negros) relativa a un punto pi (punto azul), se obtienen dos vectores V1 y V2, y se calcula el producto cruz entre ellos para obtener el vector normal n.............................34

Figura 3.3: Proceso de segmentación de la superficie plana dominante....................................36

Figura 4.1: Diagrama de clases (extracción de puntos característicos)..42

Figura 4.2: Diagrama de clases (segmentación de la superficie plana dominante)..................................43

Figura 4.3: Diagrama de flujo correspondiente al proceso de extracción de puntos característicos sobre una escena 3D...46

Figura 4.4: Diagrama de flujo del algoritmo RANSAC en una nube de puntos..........................48

Figura 4.5: Diagrama que representa el flujo de proceso del cálculo de inliers................................50

Figura 5.1: Esquema de procesamiento de nubes de puntos mediante FPGA y CPU..............................52

Figura 5.2: Esquema de división de memoria por bloques...54

Figura 5.3: Interpretación gráfica de los datos obtenidos de un arreglo 2D en un ciclo de lectura.54

Figura 5.4: Memoria de nube de puntos. 55

Figura 5.5: Diagrama de hardware de alto nivel para el proceso de extracción de puntos característicos de una nube de puntos...57

Figura 5.6: Conexiones internas el módulo producto cruz...58

Figura 5.7: Diagrama de alto nivel del módulo generador de fila y columna.................................59

Figura 5.8: Funcionamiento de la unidad de control representado en forma de carta ASM.....................61

Figura 6.1: Arquitectura simplificada de un sistema CPU+GPU..64

Figura 6.2: Sistema heterogéneo CPU+GPU conectada mediante el "puente norte"....................65

Figura 6.3: Configuración de un sistema heterogéneo CPU+GPU con múltiples CPU.........66

Figura 6.4: Configuración de un sistema heterogéneo CPU+GPU con múltiples GPU conectados al northbridge......................................67

Figura 6.5: Configuración de un sistema heterogéneo CPU+GPU con múltiples GPU encapsulados en un único chip..............................68

Figura 6.6: Configuración de un sistema heterogéneo CPU+GPU con múltiples CPU y múltiples chips Dual-GPU....................................68

Figura 6.7: Código escrito en CUDA C.................71

Figura 6.8: Diagrama representativo de los accesos a memoria ocurridos en cada iteración del cálculo de vectores normales...72

Figura 6.9: Diagrama de actividades para el flujo de ejecución (en paralelo) del proceso de extracción de características en una nube de puntos....................74

Figura 6.10: Diagrama de actividades que representa el flujo de ejecución (en paralelo) del proceso de cálculo de inliers.................................77

Figura 7.1: Comparativa del tiempo de ejecución para el proceso de extracción de puntos característicos...80

Figura 7.2: Comparativa del tiempo de ejecución para el proceso de segmentación de la superficie plana dominante..81

Figura 7.3: Representación gráfica de los vectores normales de una escena 3D...................................83

Figura 7.4: Representación RGB de puntos característicos...84

Figura A.1: Arquitectura de alto nivel de la tarjeta de desarrollo Intel DE2i-150................................92

Figura A.2: Sistema de Hardware desarrollado para la adquisición y envío de datos desde el FPGA hacia el CPU...94

Figura A.3: Diagrama de alto nivel que representa el proceso de comunicación entre el CPU y el FPGA..96

Índice de tablas

Tabla 4.1: Comparativa entre tipos de ejecución…39

Tabla 4.2: Simbología correspondiente a un diagrama de actividades..45

Tabla 5.1: Tabla de correspondencia entre coordenadas espaciales y posición en bits.............59

Tabla 7.1: Comparativa del tiempo de ejecución para el proceso de extracción de puntos característicos...80

Tabla 7.2: Comparativa del tiempo de ejecución para el proceso de segmentación de la superficie plana dominante..81

Tabla 7.3: Cálculo del ECM entre los distintos sistemas...82

Tabla 7.4: Cálculo de SSIM sobre las imágenes RGB generadas por los diferentes sistemas...........85

Tabla A.1: Tabla de estados para la transmisión de datos desde el CPU hacia el FPGA........................97

Bibliografía

LD09	Lastovetsky, A., & Dongarra, J. (2009). *High Performance Heterogeneous Computing*. New York, NY, USA: Wiley-Interscience.
Som04	Sommerville, I. (2004). *Software Engineering*. USA: Addison-Wesley.
Bha95	Bhasker, J. (1995). *A VHDL Primer*. Englewood Cliffs, NJ, USA: Prentice Hall.
Gates07	Gates, B. (2007). *A robot in every home*. Obtenido de http://www.cs.virginia.edu/~robins/A_Robot_in_Every_Home.pdf
NIAM04	Nonaka, S., Inoue, K., Arai, T., & Mae, Y. (2004). *Evaluation of Human Sense of Security for Coexisting Robots using Virtual Reality*. IEEE International Conference on Robotics & Automation. New Orleans, LA, USA.
PCL15	Point Cloud Library. (2015). *Página oficial*. Obtenido de http://pointclouds.org/
BC11	Bogdan, R., & Cousins, S. (2011). *3D is here: Point Cloud Library (PCL)*. IEEE International Conference on Robotics and Automation (ICRA). Shangai, China.
SVC10	Savage, J., Vázquez, G., & Chávez, N. (2010). *Diseño de Microprocesadores*. México: UNAM.
Sta97	Stallings, W. (1997). *Sistemas Operativos*. Madrid, España: Prentice Hall.
Sta05	Stalling, W. (2005). *Organización y Arquitectura de Computadoras*. Madrid, España: Prentice Hall.
PH08	Patterson, D., & Henessy, J. (2008). *Computer Organization and Design: The Hardware/Software Interface*. USA: Morgan Kaufmann.

X14 Xilinx Inc. (2014). *What is a FPGA?*. Obtenido de http://www.xilinx.com/fpga/index.htm

Free09 Freeman, W. (2009). *FPGA Architecture*. Obtenido de http://www.vision.caltech.edu/CNS248/Fpga/fpga1a.gif

Kall10 Kallstrom, P. (2010). *An example of how an FPGA logic cell can look like*. Obtenido de http://commons.wikimedia.org/wiki/File:FPGA_cell_example.png

GENERA15 GENERA Tecnologías. (2015). *Sistemas Embebido SoC*. Obtenido de http://www.generatecnologias.es/sistemas_embebidos_fpga.html

NVIDIA15 NVIDIA Corporation. (2015). *NVIDIA Worldwide*. Obtenido de http://www.nvidia.com/

NVIDIA09 NVIDIA (2009). *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. Obtenido de http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf

BDHHS10 Brodtkorb, A., Dyken, C., Hagen, T., Hjelmervik, J., & Storaaslli, O. (2010). *State-of-the-art in heterogeneous computing*. Scientific Programing. 18, 1-33.

Hea03 Heath, S. (2003). *Embedded systems design*. Burlington, MA, USA: Newnes.

Terasic13 Terasic Inc. (2013). *DE2i-150 FPGA Development Kit*. Obtenido de http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=11&No=529

Kjaer10 Kjaer-Nielsen, A. (2010). *Real-time vision using FPGAs, GPUs and multi-core CPUs*. Tesis doctoral publicada en línea. University of Southern Denmark, Dinamarca. Consultada en: http://covil.sdu.dk/publications/AndersPHD.pdf

HHRB11 Holz, D., Holzer, S., Rusu, R., & Behnke, S. (2011). *Real time plane segmentation using RGB-D cameras*. RoboCup International Symposium. Estambul, Turqía.

KH06 Kolman, B., & Hill, D. (2006). *Álgebra lineal*. México: Pearson Education.

WLQX12 Wang, Z., Liu, H., Qian, Y., & Xu, T. (2012). *Real-Time Plane Segmentation and Obstacle Detection of 3D Point Clouds for Indoor Scenes*. Computer Vision – ECCV 2012. Workshops and Demonstrations (Springer). 7584, 22-31.

Hyun13 Hyun, Y., Woo, K., Jeong, P., Won, L., & Myung, C. (2013). *Real-Time Plane Detection Based on Depth Map from Kinect*. 44th International Symposiun on Robotics (ISR). Seúl.

B2011 Borrmann, D., Elseberg, J., Lingemann, K., & Nuchter, A. (2011). *The 3D Hough Transform for Plane Detection in Point Clouds: A Review and a new Accumulator Design*. 3D Research. 2, 3.

FB81 Fischler, M. A., & Bolles, R. C. (1981). *Random Samples Consensus: A Paradigm for Model Fitting with Application to Image Analysis and Automated Cartography*. Comunications of the ACM. 24, 381-395.

Harris00 Harris, R. (2000). *Information Graphics: A Comprehensive Illustrated Reference*. USA: Oxford University Press.

DV12 Deepa, P., & Vasanthanayaki, C. (2012). *FPGA based efficient on-chip memory for image processing algorithms*. Microelectronics Journal. 43, 916-928.

Wilt13 Wilt, N. (2013). *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. USA: Pearson Education.

OSWSB11 Oehler, B., Stueckler J., Welle J., Schulz D., & Behnke S. (2011). *Efficient Multi-Resolution Plane Segmentation of 3D Point Clouds*. 4th International Conference on Intelligent Robotics and Applications (ICIRA). Aquisgrán, Alemania.

GFS07 Gockley, R., Forlizzi J., & Simmons, R. (2007). *Natural Person-Following Behavior for Social Robots*. 2nd ACM/IEEE International Conference on Human-Robot Interaction. Washington DC, USA.

WBSS04 Wang, Z., Bovik, A., Sheikh, H., & Simoncelli, E. (2004). *Image quality assessment: From error visibility to structural similarity*. IEEE Transactions on Image Processing. 13, 600-612.