Universidad Zaragoza
1542

Trabajo Fin de Máster

Evaluación de SLAMBench en un sistema completamente heterogéneo: CPU, GPU y FPGA

Evaluation of SLAMBench on a truly heterogeneous system: CPU, GPU and FPGA

Autor/es

Marcos Canales Mayo

Director/es

Darío Suárez Gracia y Rubén Gran Tejero

Escuela de Ingeniería y Arquitectura
2018

Repositorio de la Universidad de Zaragoza – Zaguan http://zaguan.unizar.es

[21 de septiembre de 2018 at 10:55 – classicthesis]

MARCOS CANALES MAYO

EVALUACIÓN DE SLAMBENCH EN UN SISTEMA COMPLETAMENTE HETEROGÉNEO: CPU, GPU Y FPGA

[21 de septiembre de 2018 at 10:55 – classicthesis]

EVALUACIÓN DE SLAMBENCH EN UN SISTEMA COMPLETAMENTE HETEROGÉNEO: CPU, GPU Y FPGA

MARCOS CANALES MAYO

Máster en Ingeniería Informática

Supervisado por
Darío Suárez Gracia y Rubén Gran Tejero

Grupo de Arquitectura de Computadores
Departamento de Ingeniería e Informática de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

[21 de septiembre de 2018 at 10:55 – classicthesis]

Marcos Canales Mayo: *Evaluación de SLAMBench en un sistema completamente heterogéneo: CPU, GPU y FPGA,* Máster en Ingeniería Informática, © Mayo 2017

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. Marcos Canales Mayo,

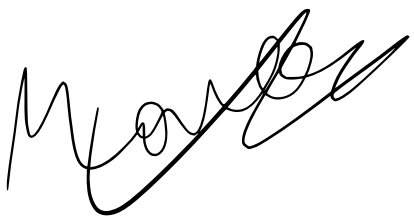
con nº de DNI 73161591-V en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Máster, (Título del Trabajo)

Evaluación de SLAMBench en un sistema completamente heterogéneo: CPU, GPU y FPGA,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 21 de Septiembre de 2018



Fdo: Marcos Canales Mayo

ABSTRACT

Existe una tendencia en los últimos años hacia usar los sistemas heterogéneos para implementar aplicaciones con requerimientos muy estrictos de rendimiento o consumo de energía, que los sistemas homogéneos no son capaces de satisfacer. En este proyecto se presenta un estudio sobre KinectFusion, una carga de trabajo significativa de visión por computador, ejecutado en un sistema 3-heterogéneo, compuesto por CPU, GPU y FPGA. Mediante el uso del benchmark SLAMBench se evalúa el impacto que tienen diversos tipos de optimizaciones sobre la implementación original de KinectFusion y enfocadas a la FPGA, pues es el dispositivo que en teoría ofrece mejor relación rendimiento/consumo energético.

En general, los resultados demuestran que el principal cuello de botella en la FPGA es la transferencia de los buffers de entrada y salida debido a las limitaciones del *hardware*. Sin embargo, aunque la GPU alcanza 3,5 veces más ancho de banda, la FPGA es un dispositivo capaz de dar un rendimiento competitivo y podría usarse como acelerador en un sistema más equilibrado. Esto es gracias a las diversas técnicas de optimización que se pueden aplicar y con las que se han conseguido, para el caso estudiado, un *speedup* de 16,54 con respecto a la implementación original. Las optimizaciones que mayor impacto han tenido están relacionadas con el alineamiento y patrón de acceso a memoria. Por el contrario, la técnica más importante que no ha dado resultados es el uso de representación en coma fija, ya que los requerimientos de precisión y el *overhead* producido por las conversiones hunden el rendimiento de la aplicación.

Este trabajo sienta las bases para una futura investigación más detallada sobre el papel que podría jugar una FPGA en entornos heterogéneos y, en concreto, sobre su posible rol en un algoritmo complejo como KFusion.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

LISTINGS

1 INTRODUCCIÓN

La evolución en la capacidad de cómputo de los procesadores de propósito general ha sido exponencial desde los años 60. Desde entonces el tamaño de los transistores se ha ido reduciendo progresivamente. Tal y como determinó Moore [15], este hecho hacía posible doblar la cantidad de transistores que contenían los *chips* en cada una de estas "iteraciones" de dos años - observación denominada ley de Moore. A su vez, ésto permitía reducir el voltaje y corriente necesario para que los transistores puedan funcionar correctamente. En línea con esto, se podía incrementar la frecuencia de reloj del *chip* (es decir, aumentar el número de instrucciones ejecutadas por unidad de tiempo) manteniendo constante la densidad de potencia (i.e. potencia disipada por unidad de superficie) entre iteraciones. Estas son las reglas de escalado que explicó Dennard con mayor detalle en su trabajo [4].

Sin embargo, desde aproximadamente el año 2005 [13], debido a ciertas limitaciones tecnológicas ya no es posible reducir en la misma medida el tamaño de los transistores, así como el voltaje y corriente que se les aplica. Esto ha provocado una deceleración en la densidad de transistores respecto a la ley de Moore, mostrado en la Figura 1. Además, el fin del escalado de Dennard conlleva que la densidad de potencia aumente desde entonces, como se puede ver en la Figura 2. Esto significa que aumentar la frecuencia de reloj (en el paso de una generación de procesadores a otra) implicaría generar más calor que los sistemas de refrigeración no serían capaces de soportar. Es por ello que, para solventar este problema, emergió el modelo del procesador *multicore*, es decir, disponer de varios procesadores a una frecuencia moderada con hilos de ejecución paralelos. Esto sustituye al modelo tradicional en el que se dispone de un único procesador a alta frecuencia.

No obstante, los avances en múltiples campos, como la visión por computador, han conducido al desarrollo de algoritmos complejos con altos requerimientos en capacidad de cómputo. En ocasiones, estos algoritmos son implementados en sistemas empotrados con limitaciones en cuanto al consumo de energía. Es el caso de las aplicaciones SLAM (*Simultaneous Localization And Mapping*), mediante las cuales un sistema autónomo es capaz de crear un mapa 3D de su entorno al mismo tiempo que se desplaza a través de él.

En este contexto, una aplicación SLAM debe cumplir con diversos requisitos:

- Ser precisa en los cálculos realizados para construir el mapa 3D.

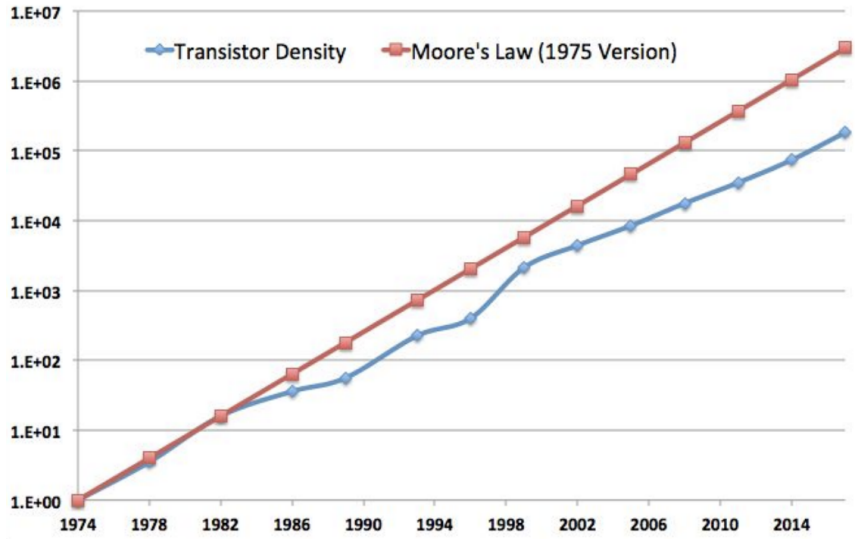


Figura 1: Deceleración de la densidad de transistores respecto a la ley de Moore - A New Golden Age for Computer Architecture [12]

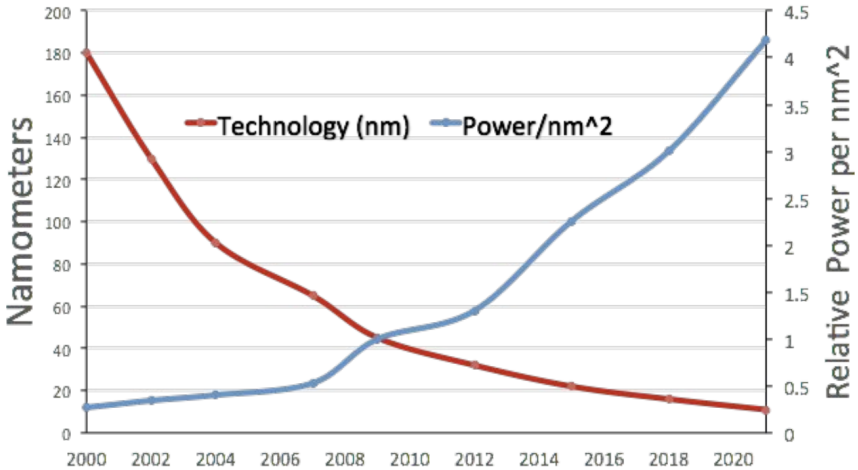


Figura 2: Evolución de la densidad de potencia vs. tecnología - A New Golden Age for Computer Architecture [12]

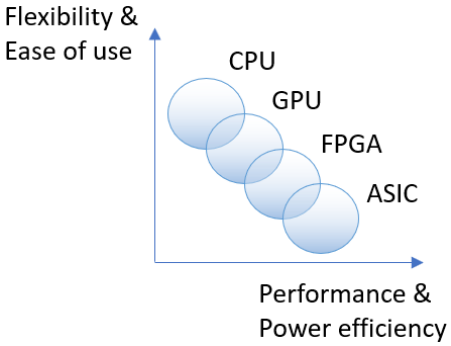


Figura 3: Flexibilidad y facilidad de uso frente a rendimiento y eficiencia energética según el tipo de dispositivo

- Garantizar un rendimiento mínimo, dadas las restricciones temporales.
- Tener un bajo consumo, debido a las limitaciones de las baterías que se usan en estos sistemas autónomos.

En muchos casos, los procesadores *multicore* (de propósito general) no satisfacen los estrictos requerimientos de este tipo de aplicaciones. La conclusión de Hennessy y Patterson a este respecto [12] es que, para afrontar este problema, es necesario usar los transistores de manera más eficiente, construyendo dispositivos especializados en vez de dispositivos de propósito general, a cambio de perder flexibilidad. Tal y como ilustra la Figura 3, por lo general cuanto más especializado sea el dispositivo mejor rendimiento y consumo energético tendrá a costa de perder flexibilidad y facilidad de uso, si bien es cierto que existen raras ocasiones en las que esto no se cumple.

A raíz de esta necesidad nace el concepto de sistemas heterogéneos, es decir, compuestos por diversos tipos de dispositivos de procesamiento especializados cada uno de ellos en realizar tareas concretas. Por ejemplo, Google diseñó en 2017 las *Tensor Processing Units* (TPUs), un acelerador específico para realizar funciones comunes en redes neuronales (inteligencia artificial), en concreto un *Application Specific Integrated Circuit*[1] (ASIC). Estos dispositivos son usados en sus propios servicios como Google Translator o el buscador de Google, además de ponerlos a disposición en sus servicios *Cloud* para el uso por cualquier persona. Esto es debido a que, como demuestran Sato [21] y Jouppi [14], tienen una relación rendimiento/consumo mucho mayor que las CPUs y GPUs más modernas, en ocasiones incluso más de dos órdenes de magnitud mayor.

1 circuito integrado para cumplir una función específica sin poder ser reprogramado

1.1 OBJETIVO

A raíz del reto de programar para sistemas heterogéneos y caracterizarlos nace SLAMBench [17], un benchmark para el popular algoritmo SLAM denominado KinectFusion (abreviado KFusion) [18], ideado por Microsoft. SLAMBench permite medir tanto la precisión como el rendimiento de KFusion. El *benchmark* proporciona una implementación de KFusion para diversos tipos de sistemas, como por ejemplo para sistemas heterogéneos. Esto es de utilidad a la hora de evaluar y optimizar KFusion, con el fin de cumplir con los requerimientos mencionados con anterioridad.

Este trabajo se enfoca concretamente en analizar y comprender el comportamiento de una carga de trabajo significativa, como SLAMBench, en sistemas heterogéneos con dispositivos de tipo FPGA (*Field-programmable gate array*). Uno de los objetivos principales es estudiar la viabilidad de este tipo de dispositivos en comparación con otros (descritos más adelante), dado que tienen mayor potencial en términos de eficiencia energética. Adicionalmente, se estudiará cualquier posible modificación que pueda hacerse sobre la implementación original con el fin de mejorar sus resultados en sistemas heterogéneos, sin alterar el comportamiento del algoritmo en sí.

1.2 ALCANCE

El alcance de este proyecto comprende los siguientes puntos:

- Comprender qué hace KFusion e identificar los posibles *hotspots*.
- Estudiar el comportamiento y la viabilidad de SLAMBench en un sistema 3-heterogéneo (i.e. tres tipos de dispositivos), poniendo especial atención en dispositivos de tipo FPGA. Esta configuración contrasta con el Estado del Arte en sistemas heterogéneos, donde prácticamente siempre se utilizan únicamente dos dispositivos.
- Estudiar el impacto de posibles optimizaciones sobre los *hotspots* en base a los criterios previamente descritos, implementarlos y analizar sus resultados.

1.3 ESTRUCTURA DEL DOCUMENTO

A partir de este punto, el documento se estructura en los siguientes capítulos:

2. **Sistemas heterogéneos: Estado del Arte**, para detallar el concepto de heterogeneidad en este contexto y, desde esta perspec-

tiva, dar a conocer el estado de los sistemas heterogéneos al comienzo de este proyecto.

3. **Trabajos relacionados** que sirvan de referencia para la temática de este proyecto. Esto incluye describir a grandes rasgos estos trabajos, así como clarificar todo aquello que sea de utilidad de cara a mejorar cualquier aspecto este proyecto.

4. **SLAMBench**: capítulo que describe los aspectos más importantes del *benchmark* con el que se trabaja durante todo el proyecto.

5. **Herramientas y metodología de trabajo**: describir las herramientas usadas a largo de este proyecto (*hardware* y *software*) para facilitar la comprensión de los resultados y la reproducibilidad de los experimentos. Así pues, también se detallan los procedimientos seguidos a la hora de realizar ciertas tareas de interés.

6. **Técnicas exploradas para optimizar kernels**: estudio de las diversas mejoras implementadas para acelerar KFusion especialmente sobre una FPGA. Esto incluye el análisis de la versión original de KFusion, así como el desarrollo y análisis de los resultados de estas optimizaciones.

7. **Conclusiones y Trabajo futuro**: utilidad de este trabajo de investigación, principalmente poniendo el foco en describir a grandes rasgos lo que aporta a lo ya existente en la literatura. Aspectos de interés que no se han llevado a cabo pero que serían de utilidad para alcanzar los objetivos planteados.

8. **Apéndices**: detalles referenciados a lo largo del documento.

2 SISTEMAS HETEROGÉNEOS: ESTADO DEL ARTE

2.1 HETEROGENEIDAD

Los sistemas heterogéneos se distinguen por estar formados por diversos dispositivos (*devices*) de distintas características (organización, arquitectura, frecuencia de reloj, rendimiento, ancho de banda, consumo energético...) con fines distintos. Además, se caracterizan por seguir el modelo *host-device*, en el que coexisten estos dispositivos funcionan como unidades de cómputo a disposición de un anfitrión (*host*), que hace de "coordinador". Esto es, el *host* es quien decide en qué momento y qué tareas manda a los *devices*. Como se puede observar en la Figura 4, existe una memoria global que comparten todos los dispositivos de la plataforma, es decir, *host* y dispositivos (*compute devices*). Cada dispositivo se divide en unidades de cómputo (*compute units*), que a su vez se divide en elementos de procesamiento (*processing elements*, PE). Además, hay zonas de memoria, denominada memoria local, que sólo son visibles para los elementos de procesamiento de una misma unidad de cómputo. Adicionalmente, cada elemento de procesamiento dispone de una memoria privada. Por último, como se puede apreciar en la imagen, también existen zonas de memoria constante usada exclusivamente y optimizada para lecturas.

Ejecutar un algoritmo complejo en un sistema de estas características explotando adecuadamente todos los dispositivos es un problema difícil. De hecho, se puede abordar siguiendo diversas estrategias, como por ejemplo minimizando el tiempo que los dispositivos están ociosos. Otra alternativa es dividir el problema en etapas y ejecutar cada una en aquel dispositivo que mejor se adapta a sus necesidades, simulando un *pipeline*. Son muchos los factores que influyen en determinar cuál es la mejor estrategia y cómo implementarla, ya que cada dispositivo tiene distintas características.

Como se ha mencionado con anterioridad, algunas aplicaciones de campos como la visión por computador o la robótica tienen requerimientos muy exigentes en cuanto a rendimiento y/o consumo. Este problema suele resolverse utilizando sistemas heterogéneos que disponen de un procesador de propósito general (denominado *Central Processing Unit*, CPU) junto a otro procesador que funciona a modo de acelerador. Este acelerador suele ser una GPU o una FPGA. No obstante, dada la complejidad que adquieren ciertos algoritmos como KFusion y la diversidad de estructuras de datos y patrones de paralelismo de los que hacen uso, algunos de estos sistemas heterogéneos no son capaces de satisfacer los requerimientos de estas aplicaciones.

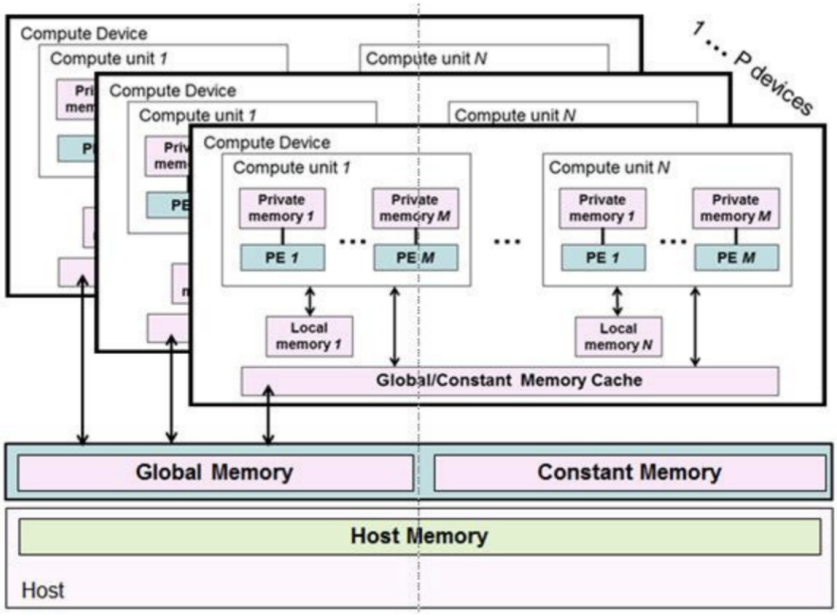


Figura 4: Modelo *host-device* - Opencl 2.2 Specification [7]

Esto es debido a que es bastante difícil adecuar la totalidad del algoritmo a un sistema 2-heterogéneo (i.e. que tiene a su disposición 2 dispositivos distintos, normalmente CPU-GPU o CPU-FPGA). Este es el principal problema que se intenta atajar en este trabajo, poniendo el foco especialmente en intentar sacar partido de la FPGA y llevando a cabo una implementación de KFusion en un sistema 3-heterogéneo compuesto de una CPU, una GPU y una FPGA.

2.2 HARDWARE

En esta sección se describe el modelo de ejecución de los dos principales aceleradores que se usan en sistemas heterogéneos: GPU y FPGA. La Figura 5 muestra un ejemplo para ambos dispositivos en el que se manda ejecutar 6 veces un *kernel* (un trabajo o función) que tiene 5 etapas. En el eje X se muestra la evolución en el tiempo, en este caso medido en ciclos de reloj del dispositivo. En cada casilla, se muestra como número el identificador de ejecución del *kernel*, mientras que la letra representa la etapa en la que se encuentra dicha ejecución. Como se puede observar, la GPU es capaz de ejecutar de forma paralela 3 ejecuciones del mismo *kernel*, que avanzan de manera simultánea en cada ciclo de reloj. Una vez han finalizado, comienzan las siguientes 3 ejecuciones del *kernel*, de nuevo, de manera paralela. En cambio, tal y como se puede observar, la FPGA funciona como un procesador segmentado, en el cual todas las instrucciones avanzan cada ciclo hacia la etapa siguiente. Este concepto es importante, dado que, igual que en las CPUs, una instrucción puede quedarse en una

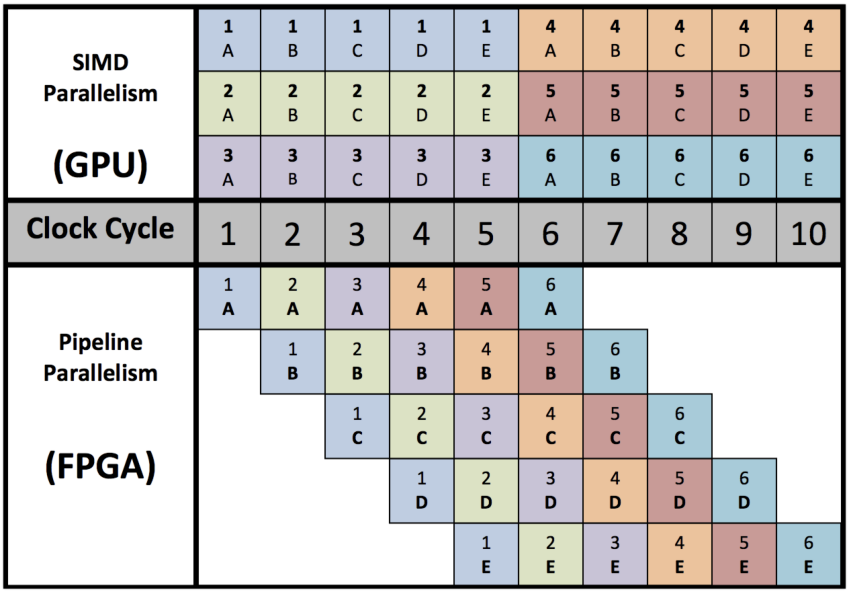


Figura 5: Comparación de paralelismos SIMD vs *Pipeline* - OpenCL on FPGAs for GPU Programmers

etapa durante varios ciclos, bloqueando el resto de instrucciones que la siguen en el *pipeline*.

2.2.1 *GPU*

Las *Graphics Processing Unit* (GPU) son aceleradores dedicados principalmente, aunque no únicamente, al procesamiento de gráficos y cálculos en coma flotante. Se caracterizan por contener un número grande de núcleos que ejecutan instrucciones de forma paralela. Debido a esta característica, son dispositivos ideales para ejecutar aplicaciones "embarazosamente paralelas" (o *embarrassingly parallel* por su traducción del inglés), es decir, aquellas en las que la carga de trabajo sea fácilmente separable en multitud de "subtrabajos" y que éstos sean poco dependientes o independientes entre sí.

Para comprender mejor la arquitectura de una GPU, formulemos el modo de calcular su *throughput*[1]. Con el fin de simplificar, vamos a obviar situaciones como esperas por operaciones de memoria, *barriers*[2], etc. Así pues, podríamos concluir con la siguiente fórmula: $\mathtt{Throughput} = \mathtt{BytesPerWorkItem} * \mathtt{Frequency}$

- **Throughput**: rendimiento medido en *bytes* por segundo.
- **BytesPerWorkItem**: trabajo en *bytes* que es capaz de producir un "trabajador".

1 rendimiento o cantidad de trabajo capaz de resolver por unidad de tiempo
2 punto de sincronización que obliga a esperar a que todos los *threads* hayan llegado este punto

- **Frequency**: frecuencia de reloj de la FPGA en hercios (ciclos por segundo).

2.2.2 *FPGA*

Las *Field-Programmable Gate Array* (FPGA) son dispositivos que contienen circuitos lógicos reprogramables. El paralelismo de este tipo de dispositivos es diferente, ya que se caracterizan por tener una estructura de *pipeline* como se puede ver en la Figura 5, similar a la arquitectura segmentada de una CPU.

En el caso ideal, el *throughput* de una FPGA podría medirse de la siguiente manera:

$$\mathtt{Throughput} = \mathtt{BytesPerWorkItem} * (\mathtt{Frequency}/\mathtt{max}(\mathtt{InitiationInterval}))$$

El **initiation interval** es el número de ciclos que tarda una iteración de un bucle en ejecutarse (cada bucle puede tener un initiation interval distinto). En la ecuación se tiene en cuenta el máximo *initiation interval* ya que es el perteneciente al bucle que retrasará las instrucciones que se encuentren en etapas anteriores del *pipeline*.

En el ámbito de la programación para FPGAs, existen los *Hardware Description Languages* (HDL), lenguajes como *VHDL* o *Verilog* que permiten programar FPGAs a bajo nivel, definiendo la estructura de los circuitos digitales y de sus señales. Aunque esto ofrece la oportunidad de afinar una aplicación con alto grado de detalle, también hay que considerar el tiempo de desarrollo, que puede aumentar de forma notable en función de la complejidad de la aplicación.

No obstante, gracias al proceso de HLS (*High Level Synthesis*), es posible generar automáticamente un diseño en código HDL para la FPGA a partir de un comportamiento programado en un lenguaje de más alto nivel como C, C++ o OpenCL. Esto facilita el desarrollo para FPGAs y abre la puerta a una mayor comunidad de desarrolladores, puesto que los HDLs tienen una curva de aprendizaje muy pronunciada.

2.3 SOFTWARE

En este apartado se justifica y describe el *software* con el que sea posible explotar las capacidades del *hardware* disponible. En este sentido y dado el alcance del proyecto, la opción idónea es OpenCL, principalmente debido a que es necesario una interfaz o lenguaje compatible con todos los dispositivos de los que se compone la plataforma heterogénea.

2.3.1 *OpenCL*

OpenCL es un *framework* que ofrece una interfaz (*Application Programming Interface*, API) basada en el lenguaje de programación C99, para programar aplicaciones que hagan uso de diversos dispositivos en una plataforma heterogénea. La mayor ventaja es que define un lenguaje común de alto nivel para varios tipos de dispositivos. Esto significa que es posible programar en OpenCL para CPU, GPU y FPGA a la vez, siempre que los fabricantes (*vendors*) de estos dispositivos provean de un *driver* de OpenCL para hacer uso de ellos. Desde este punto de vista, un caso opuesto a OpenCL es CUDA, una API también basada en C99, pero con la que únicamente es posible usar GPUs de NVIDIA.

Programar para FPGAs con OpenCL también es una ventaja en términos de tiempo de desarrollo, frente a usar directamente un HDL, ya que existen herramientas que aplican el proceso HLS mencionado anteriormente: analizan este lenguaje de alto nivel y generan el diseño *hardware* correspondiente.

En el apéndice A.1 se muestra un ejemplo de una aplicación que usa OpenCL, acompañada de comentarios para facilitar su comprensión.

2.3.2 *Altera Offline Compiler*

El compilador offline de Altera (AOC por sus siglas en inglés) es una herramienta necesaria para poder utilizar OpenCL con FPGAs de Altera, de la empresa Intel. El propósito de esta herramienta es realizar el proceso de HLS para FPGAs de este fabricante, es decir, en base al código OpenCL que se necesita ejecutar en el *device*, generar un diseño e implementación en la FPGA en términos de puertas lógicas. Esta implementación se almacena en un fichero binario que en tiempo de ejecución se carga en la FPGA.

Generar el fichero binario es una tarea que requiere un tiempo considerable para el compilador, que depende de la complejidad del código y de la capacidad en recursos de la FPGA. Para el caso que nos ocupa y dependiendo del código OpenCL compilado, el AOC puede tardar entre 45 minutos y 1 hora y media en analizar el código, crear un diseño y generar la implementación para las FPGAs usadas en este trabajo. No obstante, esta herramienta trae consigo utilidades que permiten emular el comportamiento del código OpenCL en las FPGAs. Esto facilita la tarea de desarrollo, ya que es posible **validar** el comportamiento y resultado de la aplicación sin necesidad de compilar. También es capaz de generar informes sobre el código OpenCL, sin necesidad de compilar. Estos informes sirven para analizar en detalle cómo funcionará cada porción del código OpenCL en la FPGA, con el fin de mejorar si es posible la implementación.

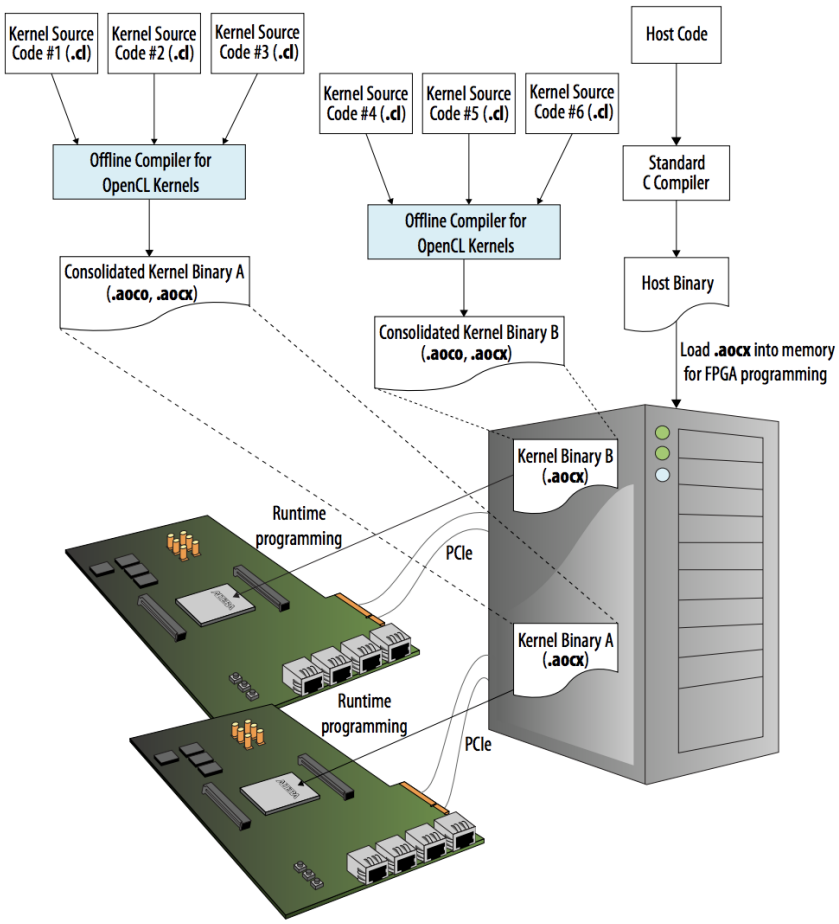


Figura 6: OpenCL FPGA Programming flow - Intel FPGA SDK for OpenCL Pro Edition Programming Guide [10]

La Figura 6 ilustra cómo se trabaja con OpenCL y el AOC para usar FPGAs de Altera. Los distintos ficheros OpenCL, que contienen todos los *kernels* que van a ejecutar los dispositivos, sirven de entrada para el AOC y su proceso de HLS. Aunque en la imagen no se puede observar, durante el proceso de compilación se generan ficheros que contienen código en un HDL con un diseño a nivel de *hardware*. Finalmente se generan un fichero binario con extensión *.aocx* y otro *.aoco* que sirven para reprogramar la FPGA en tiempo de ejecución con este diseño. Por su parte, el código C/C++ del anfitrión, haciendo uso de librerías de OpenCL, carga estos binarios durante la ejecución del programa y hace uso de las FPGAs.

3

TRABAJOS RELACIONADOS

En este capítulo se presentan trabajos importantes de ejecución desde lenguajes de alto nivel en GPU y FPGAs, con el fin de dar a conocer el contexto de los sistemas heterogéneos en general y del uso de estos dispositivos en particular. Además, en capítulos posteriores también se harán referencias a trabajos que están más íntimamente relacionados con este proyecto.

Tal y como describe Muslim [16], las GPUs ofrecen alto rendimiento en operaciones de coma flotante y gran ancho de banda en transacciones de memoria. Por el contrario, las FPGAs se caracterizan, entre otras cosas resumidas por Muslim [16] y Demirsoy [2], por ofrecer una buena relación entre el rendimiento y consumo de energía. OpenCL y el compilador *offline* de Altera son las herramientas principales usadas hoy en día por la comunidad de desarrolladores con el objetivo de alcanzar un buen ratio rendimiento/consumo en este tipo de dispositivos. Existen múltiples optimizaciones (Singh [5]) que pueden aplicarse a diversidad de aplicaciones con el fin de maximizar este ratio. Además, como ilustra Denisenko [3], la creación de un lenguaje de alto nivel como es OpenCL abre la puerta a mayor número de desarrolladores, ya que no es necesario tener un conocimiento detallado tanto del hardware como de las herramientas y flujo de trabajo tradicionales con FPGAs. Al mismo tiempo, incrementa la productividad del desarrollador al utilizar un entorno de desarrollo más familiar. Asimismo, otro de los beneficios de OpenCL es que el código es *cross-platform*[1]. No obstante, a la hora de implementar una aplicación en OpenCL, el desarrollador ha de ser consciente del tipo de dispositivo subyacente, ya que una porción de código OpenCL que da buenos resultados en una GPU no tiene por qué funcionar bien en una FPGA.

Hay muchos ejemplos del éxito de ciertas aplicaciones en FPGAs a CPUs y GPUs. Es el caso de la implementación de *Stencil*[2] llevada a cabo por Jia [11], en la que consigue una mejora de dos órdenes de magnitud en el rendimiento respecto a la versión original del algoritmo. Además, es capaz de caracterizar el sistema de memoria de la FPGA, observando su comportamiento al hacer uso de la memoria local, de las capacidades de vectorización, etc. Otro ejemplo de *Stencil* es el realizado por Waidyasooriya [24], en el que se compara su rendimiento frente a GPUs y CPUs en el ámbito de aplicaciones como el método de Jacobi, la ecuación de Laplace o el método de las Diferen-

1 el mismo código puede usarse en diferentes plataformas o entornos

2 patrón de cálculo geométrico mediante el cual se computa iterativamente el valor de un punto dados los puntos adyacentes

cias Finitas en el Dominio del Tiempo (todos usan *Stencil*). También se describen los factores que de alguna manera acotan el rendimiento en cada uno de los casos, como puede ser el límite de recursos de la FPGA o el bajo ancho de banda de acceso a memoria.

El trabajo de Tang [22] enseña los resultados de una implementación de K-means que realiza una parte del algoritmo en la CPU y otra en la FPGA, de tal manera que se maximiza el rendimiento global. Adicionalmente, compara el rendimiento y consumo de la FPGA frente a una CPU y una GPU. Sin embargo, es necesario tener en cuenta que en este caso la FPGA dispone de tecnología mucho más moderna que los otros dos dispositivos, por lo que la comparativa podría estar de alguna forma sesgada en beneficio de la FPGA.

Pu [19] hace lo mismo con el algoritmo *K-Nearest neighbors* (KNN), aunque en este caso existe equilibrio entre los tres dispositivos. De nuevo, se describen las distintas optimizaciones que han llevado a la mejor versión de KNN para la FPGA. Termina mostrando los resultados en los que se aprecia que el rendimiento de la GPU es el mayor de todos y, en concreto, tres veces mayor que el la FPGA. Sin embargo, la FPGA a su vez consigue un rendimiento por consumo 3 veces mayor que el de la GPU.

Por último, Verma [23] presenta un caso de estudio similar con el benchmark *OpenDwarfs*. Uno de los puntos interesantes que trata son algunas de las limitaciones que tiene el compilador de Altera ([1]) a la hora de aplicar heurísticas y estimar el uso de recursos, ya que esto no siempre genera la mejor configuración hardware para la FPGA. Por lo tanto, es conveniente que el desarrollador sea consciente de estas limitaciones y sepa actuar en consecuencia para dar información adicional al compilador y evitar estas situaciones.

Todos estos trabajos vienen a demostrar que las FPGAs pueden competir con las GPUs, que es el tipo de dispositivo que tradicionalmente ha abarcado gran parte del mercado de los aceleradores.

4 SLAMBENCH

Los trabajos en el campo de visión por computador suelen poner de manifiesto problemas a la hora de resolver de poner en práctica de manera eficiente algoritmos como KFusion [18]. Además, se puede observar que el foco de los autores en este sentido es siempre desarrollar una implementación adecuada para GPUs. Sin embargo, como se ha mencionado con anterioridad, es posible que existan otros aceleradores (las FPGAs en el caso que nos ocupa) que resuelvan de mejor manera las necesidades computacionales o de consumo en ciertas situaciones. Con el propósito de medir estas necesidades y aprovechar distintas formas de paralelismo en diversos tipos de dispositivos (con OpenCL, CUDA, OpenMP, ...) surge el benchmark SLAMBench [17]. Este benchmark está preparado para ejecutarse con el *dataset* ICL-NUIM [8] como entrada de datos, ampliamente utilizado en benchmarks de aplicaciones SLAM.

El trabajo realizado por Sergio Iannace [9], basado en SLAMBench e íntimamente relacionado con este proyecto, presenta determinadas optimizaciones sobre las etapas de KFusion, usando OpenCL para en FPGAs. En este contexto, justifica que puede ser idóneo hacer uso de FPGAs en ciertas situaciones, frente al uso tradicional de GPUs. Esto sirve como fundamento para el comienzo de este proyecto y es una de las principales referencias durante el desarrollo del mismo.

4.1 KINECTFUSION

KFusion es un algoritmo que, a partir de los *frames* o imágenes tomadas sucesivamente por una cámara, calcula la trayectoria de esta cámara en el espacio y construye un volumen 3D. El algoritmo produce un resultado como el que aparecen en la Figura 7. En la fila superior se puede observar la evolución de la trayectoria de la cámara conforme se mueve en torno a un punto, mientras que en la fila inferior se aprecia el volumen 3D obtenido a partir de las imágenes. La primera columna corresponde al resultado de procesar dos tercios de la vuelta alrededor del punto. En la última columna se muestran los resultados correspondientes a una vuelta entera.

La implementación de KFusion proporcionada en SLAMBench se divide en las etapas mostradas en la Figura 8 (ejecutadas para cada imagen). Cada círculo corresponde a un kernel y su color indica la etapa a la que pertenece. Las flechas y sus etiquetas indican una dependencia entre kernels, siendo el origen de la flecha el productor y

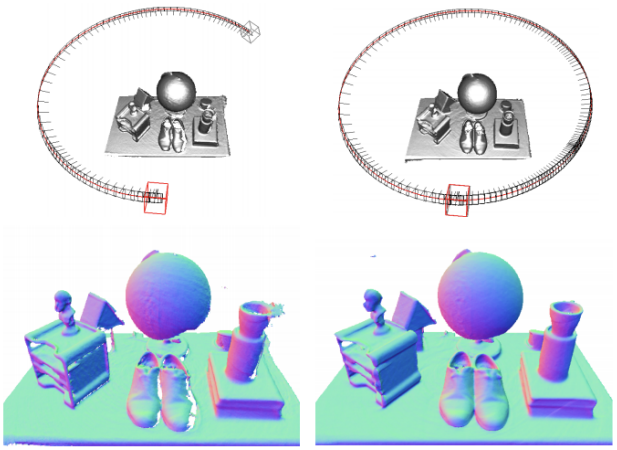


Figura 7: Trayectoria y volumen calculados por KFusion a lo largo del tiempo - KinectFusion: Real-Time Dense Surface Mapping and Tracking [18]

el destinatario el consumidor. En resumen, estas son las etapas en las que se divide:

1. **Preprocesado de la imagen tomada** (color verde): se aplica un filtro bilateral para mejorar la detección de puntos de interés en etapas posteriores. El funcionamiento de este filtro se explica en el Apéndice A.2.

2. **Tracking** (color azul): detección de puntos característicos (esquinas, bordes...) y estimación de la nueva pose de la cámara. Esto es posible gracias a la correspondencia que es capaz de establecer entre puntos de una imagen y puntos de las anteriores imágenes. Sin aplicar el filtro bilateral esta etapa perdería precisión.

3. **Integrate** (color naranja): integración de los puntos característicos en la nube o volumen de puntos 3D. Este volumen es la estructura 3D que se va construyendo mientras se van tomando las imágenes.

4. **Raycast** (color rojo): traza rayos a los puntos para conocer si la superficie es visible o no.

5. **Rendering** (color amarillo): renderiza el volumen obtenido.

4.2 BENCHMARK

SLAMBench es un benchmark que ofrece la posibilidad de evaluar el comportamiento de KFusion en diversas implementaciones:

- Secuencial en C++: utilizando un hilo de ejecución de la CPU.

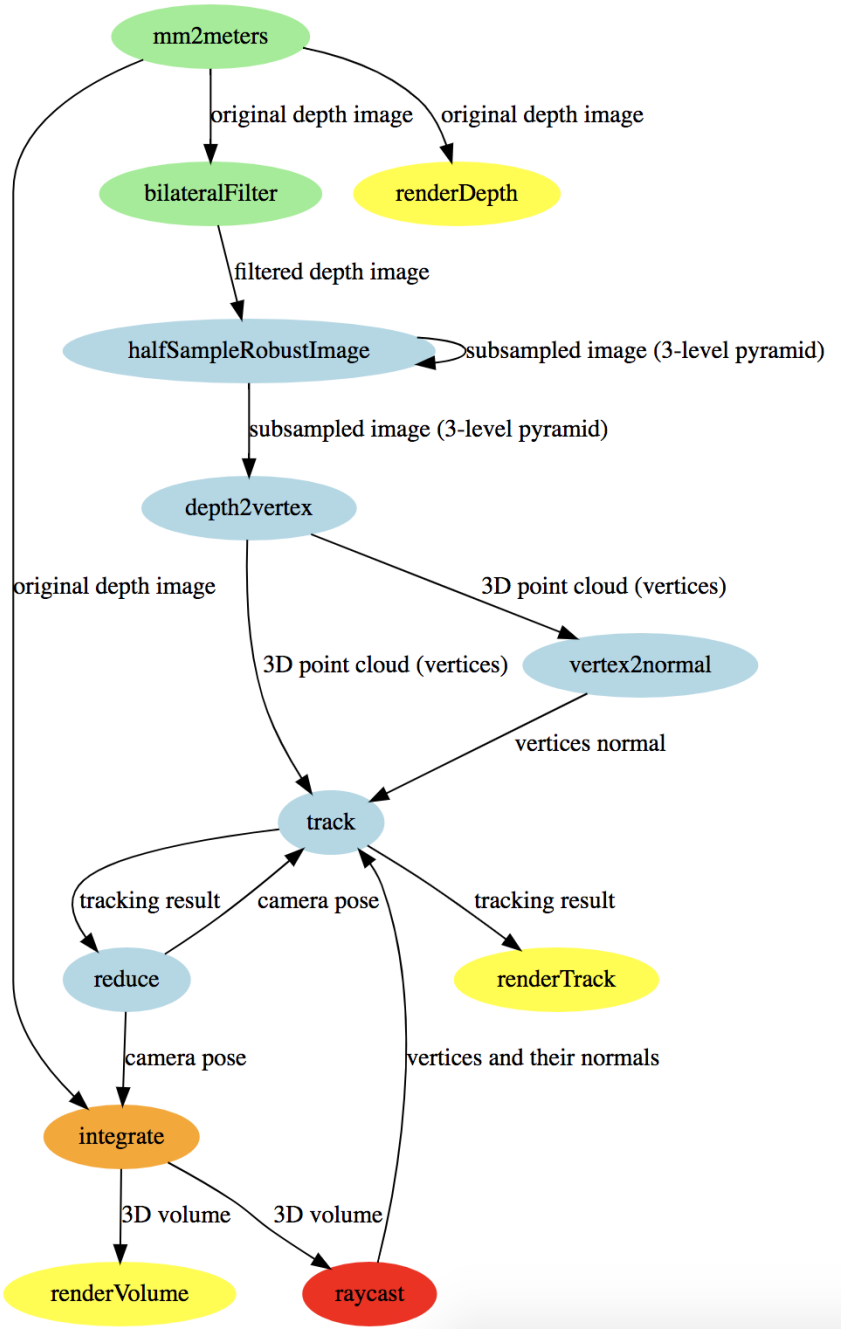


Figura 8: Etapas en que las se divide KFusion

- OpenMP: aprovechando las capacidades *multicore* de la CPU.
- CUDA: usando únicamente GPUs de NVidia.
- OpenCL: usando únicamente un acelerador cualquiera compatible con OpenCL.

Uno de los objetivos principales de este proyecto es analizar e implementar en la FPGA aquellas partes del algoritmo KFusion que puedan ajustarse a las características de la FPGA, sin perder los beneficios que proporcionan tanto la CPU como la GPU. Esta es una de las razones por las que se escoge OpenCL como *framework* de trabajo en este proyecto, y que se detalla en el capítulo correspondiente.

5 HERRAMIENTAS Y METODOLOGÍA DE TRABAJO

5.1 PLATAFORMAS DISPONIBLES

Con el fin de facilitar la comprensión de los tiempos obtenidos durante los experimentos y facilitar su reproducibilidad, en la Tabla 1 se presentan las características más relevantes de las plataformas de las que se disponen en la Tabla 1. Cabe destacar que la plataforma Socarrat es una estación de trabajo con una GPU y una FPGA discretas, mientras que la plataforma DE1-SoC integra una CPU y una FPGA en el mismo *chip*.

Tabla 1: Plataformas disponibles durante el proyecto

Plataforma	CPU			
	GHz	Núcleos	Threads	Memoria[1]
Socarrat	4,2	4	8	64 GB
DE1-SoC	2	2	2	1 GB
	GPU			
	GHz	Núcleos	Conector	Memoria[2]
Socarrat	1	3072	PCIe 3.0 x16[3]	12 GB
DE1-SoC	N/A	N/A	N/A	N/A
	FPGA			
	On/Out Chip		Conector	Memoria[2]
Socarrat	Out of chip		PCIe 3.0 x8[4]	4 GB
DE1-SoC	On of chip		N/A	64 MB

[1] memoria global de la plataforma
[2] memoria integrada en el dispositivo
[3] velocidad máxima de 15,76 GB/s
[4] velocidad máxima de hasta 7,88 GB/s

Teniendo estas características en cuenta y que sería interesante experimentar con un sistema 3-heterogéneo, la principal plataforma de trabajo y donde se realizarán la mayoría de experimentos es **Socarrat**. Sin embargo, se ha de tener en cuenta que trabajar con ésta tiene los siguientes inconvenientes:

- Es compartida entre varios investigadores, por lo que los experimentos pueden ocasionalmente estar "contaminados". No obs-

[21 de septiembre de 2018 at 10:55 – classicthesis]

tante, para evitar esto, antes de realizar cualquier experimento se observará si el sistema está siendo usado por otra persona. Además, se realizará un número de ejecuciones determinado para cada experimento, siendo el resultado válido la media de todos ellos. También se pondrá especial atención al coeficiente de variación, que servirá como medida de la validez de los experimentos.

- No es un sistema del todo equilibrado, ya que, tal y como se verá en capítulos posteriores, la FPGA está muy por detrás en capacidad de cómputo.

La totalidad de los experimentos presentados en este trabajo corresponden a ejecuciones con la versión 17.0 del AOC y del *runtime* de OpenCL de Altera, excepto para la plataforma DE1SoC que sólo dispone de la versión 16.0 del *runtime*. Los tiempos resultantes mostrados en este documento están en milisegundos, a no ser que se mencione lo contrario. En todos los experimentos se utiliza el *dataset* ICL-NUIM 2 como entrada de datos. Dado que la ejecución con el *dataset* entero puede tardar varias horas (contiene 882 *frames*), todas las medidas corresponden a la media de 20 ejecuciones con los primeros 30 *frames* del *dataset* como entrada, que consideramos representativos. Además, las medidas mostradas para cada kernel corresponden a la ejecución de únicamente ese kernel en el dispositivo determinado, mientras que el resto de kernels son ejecutados en el *host*. La razón detrás de esto es que así se puede ver cuánto se tarda en la transferencia de datos de entrada y salida entre *host* y *device* para cada kernel, ya que es algo que también hay que valorar para una implementación en un sistema heterogéneo, donde no todos los kernels se ejecutan en el mismo dispositivo. Si se ejecutaran todos los kernels en, por ejemplo, la GPU, únicamente habría transferencia de datos al inicio (envío de los datos de entrada desde el *host* a la GPU) y al final del algoritmo (envío de los resultados desde la GPU al *host*).

5.2 FLUJO DE TRABAJO CON LA FPGA

Gracias a las guías de mejores prácticas que proporciona Intel [1], se han tenido en cuenta la gran mayoría de herramientas proporcionadas para desarrollar la versión óptima de un kernel. A partir de estas guías, se ha definido el proceso mostrado en el diagrama de flujo de la Figura 9, cuyos pasos son los siguientes:

1. Implementación del kernel.
2. Emulación del kernel para comprobar la validez de la implementación en cuanto a resultados.
3. Análisis del *informe* generado (ver Apéndice A.3). Si es posible mejorar el kernel entonces se vuelve al primer paso.

Figura 9: Flujo de trabajo con la FPGA

4. Generación de los ficheros binarios .aocx y .aoco.
5. Ejecución de la aplicación.
6. Análisis de las medidas obtenidas. Si no son aceptables se vuelve al primer paso con una implementación alternativa.

La mayor diferencia respecto al flujo de trabajo propuesto por Intel, es que en este caso no se hace uso del *profiler* del *Software Development Kit* (SDK) de Altera. Esta herramienta proporciona información muy detallada, por ejemplo sobre latencia de accesos a memoria, cuánto le cuesta a la FPGA ejecutar cada línea de código, etc. En definitiva, la herramienta se encarga de recolectar todos los datos con alto grado de detalle y finalmente obtener estadísticas relevantes sobre el comportamiento del dispositivo en tiempo de ejecución. La gran desventaja y principal razón por la que se prescinde de esta herramienta es que el tiempo de ejecución del programa se ve sensiblemente afectado, pues la permanente comunicación entre *host* y FPGA lo ralentiza notablemente. Es por ello que, en vez de usar esta herramienta, se ha modificado el *framework* de SLAMBench para crear un *profiler* propio y simple utilizando funciones de librerías estándar de C y de la librería de OpenCL. Así pues, se pueden obtener métricas similares con apenas impacto en el tiempo de ejecución.

6 TÉCNICAS EXPLORADAS PARA OPTIMIZAR KERNELS

En este capítulo se explican las distintas técnicas exploradas para optimizar kernels y se analizan sus resultados en comparación con la implementación original. Aclarar que, como se verá en el apartado final, las técnicas se han aplicado de manera incremental, es decir, cada vez que se prueba una nueva técnica se "crea" una versión del código con esa y todas las anteriores que han mejorado el rendimiento de la aplicación. De esta forma la última versión tendrá todas las técnicas que han funcionado satisfactoriamente. Todas las optimizaciones que se han llevado a cabo y que se presentan en este capítulo están disponibles en un repositorio público [20].

6.1 CARACTERIZACIÓN DE KFUSION

Con el fin de comprender el funcionamiento de cada uno de los kernels de KFusion en términos de rendimiento y consumo de recursos se ha ejecutado la versión original de OpenCL del *benchmark*. Así pues, los resultados presentados en la Tabla 2 corresponden a los tiempos de ejecución de cada kernel en cada dispositivo. Asimismo, en la Tabla 3 se muestran el resultado de la suma del tiempo de transferencia de datos (*input/output*, IO) y del tiempo de ejecución del kernel en el dispositivo. En la Figura 10, se puede comparar para cada kernel la cantidad de tiempo que tarda la FPGA de Socarrat en transferir datos y ejecutar el propio kernel. La Figura 11 muestra los mismos datos para la GPU de Socarrat. Teniendo en cuenta estas dos gráficas, la diferencia entre el tiempo de IO y el de los propios kernels es notable en ambos FPGA y GPU.

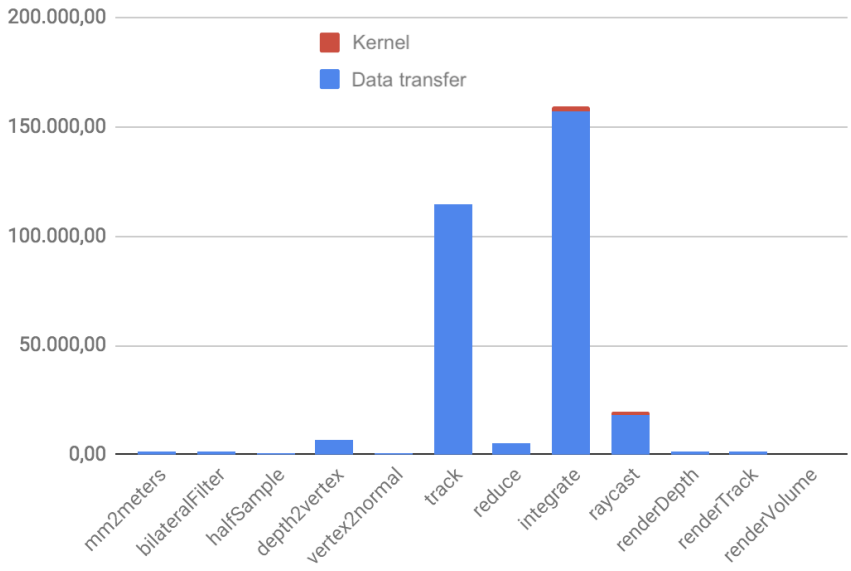


Figura 10: Tiempos (ms) de la versión original del código de cada kernel OpenCL en la FPGA de Socarrat

Tabla 2: Tiempo (ms) de ejecución de kernel en la versión original de OpenCL

Etapa	Kernel	Socarrat			DE1-SoC	
		CPU	FPGA	GPU	CPU	FPGA
prepro	mm2meters	8,3	9	0,3	75,5	19
	bilateralFilter	715,2	201,4	4,8	13131	414
track	halfSample	2,7	12,9	0,5	25	77,7
	depth2vertex	13,4	15,3	1	196,7	36,8
	vertex2normal	21,1	23,9	1,6	444,5	172,9
	track	238,9	74,5	11,7	4955,5	426,6
	reduce	73	98,7	28,4	1411,9	NA[1]
integrate	integrate	1115,6	1.728	6	19044	2734
raycast	raycast	3157	1689,4	6,9	24030,1	NA[1]
render	renderDepth	10,8	11,8	0,4	205,8	22,3
	renderTrack	4,8	14,6	0,5	229,9	43,3
	renderVolume	938,6	NA[1]	2	9003,1	NA[1]

[1] la FPGA no dispone de suficientes recursos para implementar el kernel

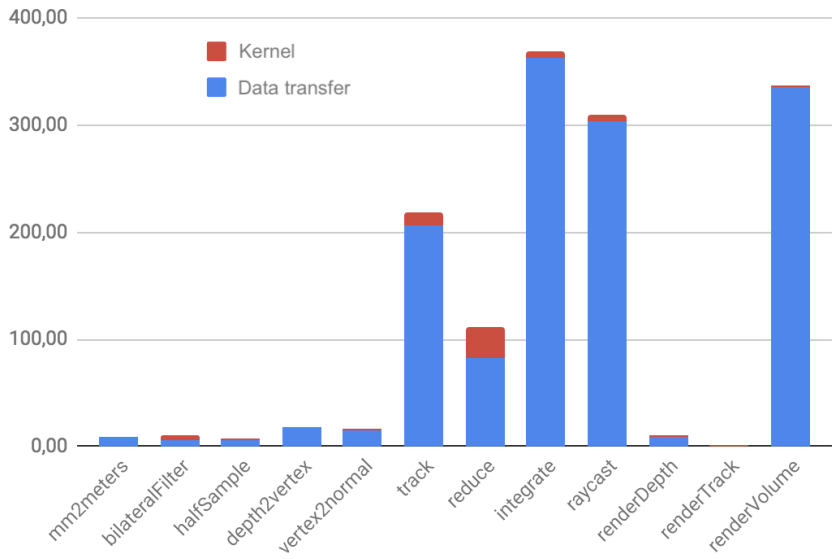


Figura 11: Tiempos (ms) de la versión original del código de cada kernel OpenCL en la GPU de Socarrat

Tabla 3: Tiempo total (ms) en la versión original de OpenCL

Etapa	Kernel	Socarrat			DE1-SoC	
		CPU	FPGA	GPU	CPU	FPGA
prepro	mm2meters	8,3	1332,9	8,8	75,5	364,4
	bilateralFilter	715,2	1468,7	11,1	13131	657,4
track	halfSample	2,7	452	6,6	25	228,2
	depth2vertex	13,4	6381,3	18,69	196,7	149,2
	vertex2normal	21,1	345,7	16,04	444,5	665,9
	track	238,9	114910,8	219	4955,5	24767,4
	reduce	73	5521,5	111,5	1411,9	NA[1]
integrate	integrate	1115,6	159111,2	368,6	19044	55826,4
raycast	raycast	3157	19830,6	309,8	24030,1	NA[1]
render	renderDepth	10,8	1279,1	9,7	205,8	270,7
	renderTrack	4,8	1694	0,6	229,9	987,8
	renderVolume	938,6	NA[1]	337,4	9003,1	NA[1]

[1] la FPGA no dispone de suficientes recursos para implementar el kernel

Cabe destacar que la GPU es el dispositivo más rápido en todos los kernels en términos de ejecución, llegando a ser dos órdenes de magnitud más rápida que la CPU y la FPGA en los kernels de *integrate* y *raycast*, que son los más lentos en estos dos dispositivos. No obstante,

si se suma el tiempo de IO la CPU es más rápida en los kernels de *mm2meters*, *halfSample*, *depth2vertex* y *reduce*. Esto es debido a que los cálculos no son lo suficientemente intensivos como para que merezca la pena que la GPU los resuelva, ya que el tiempo de IO entre *host* y GPU termina ocultando el tiempo de ejecución del kernel. Aún así, la suma de estos kernels apenas suponen un 6 % del tiempo de ejecución en la GPU, por lo que no merece la pena poner el foco en ellos, ya que cualquier optimización que se les aplique tendría un impacto muy pequeño en el tiempo global.

En general, se puede apreciar que los kernels que más tiempo consumen tanto en cálculo como en IO son *track*, *reduce*, *integrate* y *raycast*. Excepto *reduce* (ver código original en el Apéndice A.5), todos ellos tienen un alto "grado" de divergencia, es decir, existen multitud de caminos que los *threads* pueden tomar, debido mayormente a las cláusulas *if-else*. Si un *thread* toma un camino distinto al siguiente y ambos tardan tiempos distintos en sus caminos, llegará un momento en el que uno de ellos esté esperando al otro para converger (ver implementación del kernel de *integrate* como ejemplo). Este comportamiento es difícilmente evitable y es una de las principales razones por las que este proyecto se enfocará en optimizar el kernel de *reduce* para la FPGA. Además, en la Figura 12, donde se muestra para 30 *frames* la duración de cada etapa ejecutando todo el *pipeline* en la GPU, se puede observar que este kernel es el más costoso en cuanto a tiempo de ejecución (supone más del 40 % del tiempo de ejecución total del algoritmo). Finalmente, dado que implementa el patrón *reduce* (necesita muchos datos de entrada pero produce muy pocos de salida), es uno de los *kernels* que menos transferencia de datos necesita. Esto es importante, pues el tiempo de transferencia de datos supone una penalización importante a la hora de enviar tareas a los dispositivos y la FPGA en este kernel alcanza un ancho de banda total 3,5 veces menor que la GPU (ver Apéndice A.6). Es por ello que este kernel es el candidato ideal para ser optimizado.

Adicionalmente, en el Apéndice A.4 se explican los principales tipos de recursos de los que disponen las FPGAs y el porcentaje de uso de estos recursos en la versión original en ambas FPGAs.

6.2 TÉCNICAS EXPLORADAS

6.2.1 *Memoria privada, local y global*

En el modelo *host-device* de OpenCL existen tres tipos de memoria:

- **Global**: visible por el *host* y todos los dispositivos. Suele ser grande pero lejana, por lo que el acceso a esta memoria es costoso.

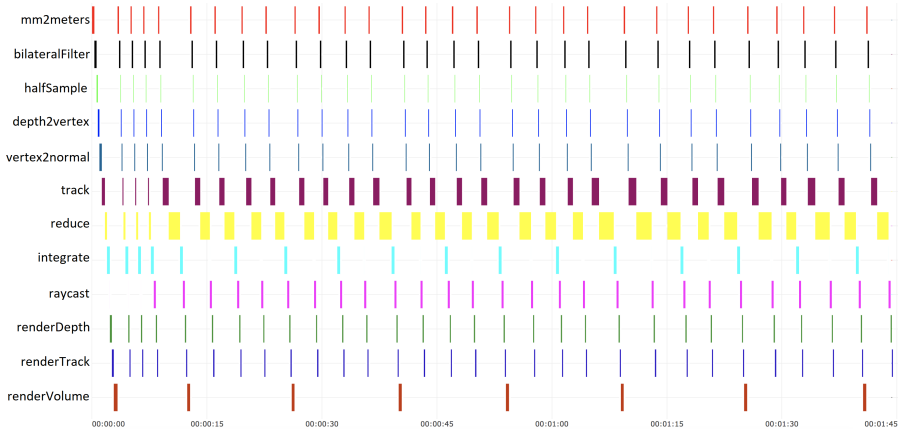


Figura 12: Duración de cada etapa ejecutando todo el *pipeline* de la versión original de OpenCL en la GPU

- **Local**: dentro del kernel, visible por todos los *work items* que pertenezcan a un mismo *work group*. Es más pequeña que la global, pero tiene menor latencia de acceso.
- **Privada**: dentro del kernel, visible sólo por un *work item*. Normalmente se sitúa en los registros del dispositivo, lo que significa que está a una distancia física muy corta (baja latencia), aunque son más limitados en cantidad. Esta es la memoria que se usa por defecto al declarar variables en el kernel.

Teniendo en cuenta estas características, es conveniente minimizar siempre los accesos a memoria global y usar siempre que sea posible la memoria privada. La memoria local podría usarse como cache de un grupo de *work items*. Un ejemplo de uso es la propia implementación original del kernel de *reduce* (ver Apéndice A.5). En esta implementación, cada work item resuelve un problema parcial y deja su resultado en memoria local. Finalmente, uno de los *work items* hace una reducción sobre todos estos cálculos parciales para obtener la solución al problema global.

Para definir una variable como memoria global o local es necesario añadir el modificador *__global* o *__local* al declarar la variable, como se mostrará en los ejemplos de las siguientes técnicas.

6.2.2 *Parámetros attribute*

En OpenCL es posible añadir al código directivas que proporcionen cierta información al compilador a la hora de asignar recursos a un kernel. Estas directivas son denominadas "atributos". Algunos de los atributos que se han aplicado en la medida de lo posible son los siguientes:

- **REQD_WORK_GROUP_SIZE**: Indica exactamente cuántos *work items* van a ser invocados para ejecutar un determinado kernel.

- **MAX_WORK_GROUP_SIZE**: Indica cuántos *work items* como máximo van a ser invocados para ejecutar un determinado kernel.
- **NUM_SIMD_WORK_ITEMS**: Con esto se aprovecha la capacidad de vectorización del dispositivo (*Single Instruction Multiple Data*, SIMD). Por ejemplo, si este atributo tiene valor "4" y los accesos a memoria por parte de los *work items* son consecutivos, entonces el compilador será capaz de vectorizar el acceso a memoria de manera que ejecutará una instrucción para cargar los datos correspondientes a 4 *work items*.

La siguiente porción de código muestra un ejemplo de uso de esta técnica:

```
__attribute__((reqd_work_group_size(64,1,1)))
__attribute__((num_simd_work_items(4)))
__kernel void vectsum(
        __global const float * restrict a,
        __global const float * restrict b,
        __global float * restrict c
) {
        size_t threadId = get_global_id(0);
        c[threadId] = a[threadId] + b[threadId];
}
```

Esto indicará al compilador que realice operaciones vectoriales con un tamaño de 4 *floats*. Además, el compilador sabe que siempre se lanzarán 64 *work items* para este kernel. No obstante, en los experimentos realizados el uso de recursos de la FPGA no varía en función de estos parámetros, además de que no han resultado en una mejora sobre el rendimiento de la aplicación, por lo que es posible que el compilador haya ignorado estos atributos en el caso que nos ocupa.

6.2.3 *Loop unrolling y loop pipelining*

Loop unrolling consiste en, como su nombre indica, desenrollar iteraciones de un bucle, esto es, clonar el cuerpo del bucle tantas veces como indique el grado de desenrrollado y ajustar el índice del bucle. Una de las ventajas de esto es que se reduce la lógica de control tantas veces como el grado de desenrollado. Suele combinarse con *loop pipelining*, de manera que si un bucle sigue el patrón LOAD-COMPUTE-STORE, al desenrollarlo 2 veces quedará como LOAD-LOAD-COMPUTE-COMPUTE-STORE-STORE. Así se consigue mayor rendimiento, ya que las latencias de memoria quedan ocultas. Es tarea del desarrollador definir el desenrollado de un bucle y es el AOC el que se encarga de crear la estructura de *pipeline* si es posible. Para desenrollar un bucle, han de cumplirse las siguientes condiciones:

- No debe haber dependencias entre iteraciones.
- El número de iteraciones debe ser conocido en tiempo de compilación.
- El dispositivo debe disponer de los recursos necesarios para clonar el cuerpo del bucle.

Un ejemplo de uso de *loop unrolling* es la siguiente porción de código:

```
#pragma unroll 4
for(int i=0; i<8; i++) {
        sum += a[i]*b[i];
}
```

Esto sería traducido por el compilador a un código equivalente al siguiente:

```
for(int i=0; i<2; i+=4) {
        sum += a[i]*b[i];
        sum += a[i+1]*b[i+1];
        sum += a[i+2]*b[i+2];
        sum += a[i+3]*b[i+3];
}
```

Además, siguiendo el ejemplo anterior, si el compilador crea una estructura de *pipeline* entonces el resultado puede guardarse en el propio *pipeline* hasta el final del bucle, sin necesidad de entregar los datos a memoria en cada ciclo. Esto significa que los resultados de *sum* se acumularán en el *pipeline* hasta que termine la iteración, momento en el que se entregarán a la memoria global.

Como es de esperar, el uso de recursos aumentará en función del grado de desenrollado de los bucles. En el caso del kernel de *reduce*, el uso total de recursos en la versión original en la FPGA de Socarrat es de 33 %. Sin embargo, si se desenrrollaran todos los bucles que cumplen las condiciones anteriormente mencionadas, este dato aumenta al 39 %. Aún así la mejora es inapreciable, probablemente debido a que estos bucles no tienen casi impacto en el tiempo de ejecución de este kernel.

6.2.4 *Patrón de acceso a memoria*

Los patrones de acceso a memoria también son importantes a la hora de realizar optimizaciones. Tener un patrón de acceso aleatorio podría hundir el rendimiento de un aplicación, ya que no se aprovechan los mecanismos de *cache*. Por ello, siempre es ideal tener un patrón de acceso a posiciones de memoria consecutivas, que aproveche la localidad espacial o temporal de los datos para facilitar su reuso.

En este contexto surge el concepto de memoria coalescente. Cuando se realiza una operación de lectura de un dato, la memoria no

entrega únicamente ese, sino que entrega un bloque de datos consecutivos entre los que se encuentra el pedido. Es por ello que, si *work items* consecutivos piden datos que están contiguos en memoria, el rendimiento de la aplicación mejorará notablemente, ya que siempre se aprovecharán todos los datos que la memoria entrega. Además, se pueden combinar múltiples instrucciones de memoria en una única instrucción vectorial (SIMD).

A continuación se muestra un ejemplo de un acceso a memoria aleatorio frente a uno consecutivo:

```
// Unknown random memory access
for (int i=0; i<32; i++) {
        int idx = a[i] + b[i];
        c[idx] = c[idx]*d[idx];
}
// Consecutive memory access which can be coalesced
for (int i=0; i<32; i++) {
        c[i] = a[i] * b[i];
}
```

El mejor patrón de acceso a memoria probado ha mostrado un *speedup* en el tiempo de ejecución del kernel de 1,22 en relación a la implementación original, principalmente a causa del acceso a memoria de forma coalescente mediante el cual siempre se aprovecha todo el bloque de datos leído de memoria. Por otro lado, en el tiempo de IO se ha conseguido un *speedup* de 1,08, gracias a que se ha reducido el tamaño del buffer de salida del kernel, aunque el tamaño del buffer de entrada sigue siendo considerablemente mayor. Esto es lógico, debido al propio patrón *reduce*.

6.2.5 *Memoria alineada*

Las llamadas al sistema como *malloc* o *calloc*, no reservan memoria de forma alineada, lo que supone una penalización en el tiempo de IO en el caso de las FPGAs de Altera debido a restricciones de su *runtime*. Es por ello que en estos casos, el *runtime* muestra mensajes como éste: *** WARNING: [aclo] NOT using DMA to transfer 2457600 bytes from device to host because of lack of alignment** host ptr (0x7fadc1a6d010) and/or dev offset (0x4e5800) is not aligned to 64 bytes*

Para evitar esto es posible reservar memoria de forma alineada con la llamada *posix_memalign* en el código del *host* tal y como se muestra en el siguiente ejemplo:

```
// Unaligned:
// myHostBuffer = (float*) malloc(32 * sizeof(float));
// Aligned to 64 bytes:
posix_memalign((void **) &myHostBuffer, 64, 32 * sizeof(float));
```

Únicamente haciendo este cambio sobre para los buffers del kernel de *reduce* el tiempo de IO se redujo un 95 % para la FPGA, aunque para la GPU esta técnica no tiene ningún efecto.

6.2.6 *Buffers de lectura/escritura*

La API de OpenCL permite indicar si un buffer va a ser utilizado como sólo lectura, escritura o ambos. Esto proporciona información al compilador del host y al *driver* del device sobre cómo va a ser utilizado ese buffer y por lo tanto podría ser capaz de hacer ciertas optimizaciones. Por ejemplo:

- Si un buffer está marcado como "sólo lectura" el dispositivo podría situarlo en una zona de memoria constante, ya que tiene un acceso de lectura mucho más rápido que la memoria RAM.
- Si un buffer es "sólo escritura", el *driver* del dispositivo podría decidir implementar un modelo de memoria *write-around*, es decir, todas las operaciones de memoria van a memoria principal sin pasar por *cache*. De esta forma, las operaciones sobre este buffer no ocuparán entradas en la *cache*, que podrá disponer de más espacio para otros buffers sobre los que sí se ejecuten operaciones de lectura.

A continuación se muestra un breve ejemplo de uso:

```
// Host code when creating the OpenCL buffer:
myHostOpenCLWriteOnlyBuffer = clCreateBuffer(context,
    CL_MEM_WRITE_ONLY, 32 * sizeof(float), NULL, NULL);
myHostOpenCLReadOnlyBuffer = clCreateBuffer(context,
    CL_MEM_READ_ONLY, 32 * sizeof(float), NULL, NULL);
// Device code when defining the function parameters:
__kernel void myDeviceKernel (
        __global __write_only float * myDeviceWriteOnlyBuffer,
        __global __read_only const float * myDeviceReadOnlyBuffer
) {
        // ...
}
```

No obstante, depende del *driver* del dispositivo tener en cuenta estos *flags* para realizar las optimizaciones oportunas. De hecho, estas técnicas no han supuesto ninguna mejora en el rendimiento o en el uso de recursos de la FPGA.

6.2.7 *Funciones built-in*

La especificación de OpenCL incluye un conjunto de instrucciones para ejecutar funciones que son frecuentemente utilizadas, de tal manera que el fabricante pueda proporcionar una implementación óptima para su plataforma. Es por ello que es recomendable utilizar siempre que sea posible las funciones *built-in* correspondientes, en lugar de hacer una implementación propia. A continuación se listan algunas de estas funciones (ver Apéndice A.7 para una explicación más detallada acompañada de un ejemplo de uso):

- **mad**: calcula el valor aproximado de $a \times b + c$.
- **ceil**: redondea hacia arriba un número flotante.
- **sin**: computa el seno de un número.
- **fmod**: devuelve el módulo de un número.

La siguiente porción de código muestra un ejemplo de uso de la función *mad*:

```
__kernel void myDeviceKernel (
        __global float * a,
        __global float * b,
        __global float * c
) {
        size_t threadId = get_global_id(0);
    // Using normal arithmetics
        //a[threadId] += b[threadId] * c[threadId];
    // Using built-in functions
    a[threadId] = mad(b[threadId], c[threadId], a[threadId]);
}
```

Sin embargo, para las funciones usadas en los experimentos se ha comprobado que el AOC no realiza ninguna optimización específica para el dispositivo, por lo que no se ha observado que la precisión de la aplicación o el rendimiento varíen usándose esta técnica.

6.2.8 *División en múltiples kernels*

Durante la fase de *tracking* se aplica un submuestreo, obteniendo la misma imagen en tres tamaños distintos: 320x240, 160x120 y 80x60. Esta representación de la imagen se denomina "pirámide de tres niveles". En el caso del kernel de *reduce*, siempre es lanzado con tres cantidades de *work item* distintas (una para cada nivel de la pirámide), por lo que dividirlo en tres kernels podría ayudar al compilador a realizar una mejor asignación de recursos. Esto es debido a que, para cada uno de los kernels, el número de *work items* que van a ejecutarlo es conocido en tiempo de compilación, así como el número de iteraciones de todos los bucles.

```
// Size is unknown for the compiler, but we know the application
    always passes three sizes
__kernel void myDeviceKernel (
        __global float * a,
        __global float * b,
        __global float * c,
    int size
) {
        for (int i=0; i<size; i++) {
        // ...
    }
```

```
}

// Known size for all kernels, hence better resources assignment
    and possibility to unroll and pipeline the loop
__kernel void myDeviceKernel1 (
        __global float * a,
        __global float * b,
        __global float * c
) {
        for (int i=0; i<320*240; i++) {
        // ...
    }
}

__kernel void myDeviceKernel2 (
        __global float * a,
        __global float * b,
        __global float * c
) {
        for (int i=0; i<160*120; i++) {
        // ...
    }
}

__kernel void myDeviceKernel3 (
        __global float * a,
        __global float * b,
        __global float * c
) {
        for (int i=0; i<80*60; i++) {
        // ...
    }
}
```

Gracias a esta técnica, se ha obtenido un *speedup* de 2,07 en el tiempo de ejecución del kernel, a cambio de usar un 19 % más de recursos de la FPGA.

6.2.9 *Opciones de compilación*

La especificación de OpenCL indica que ciertas opciones de compilación que pueden pasarse en tiempo de ejecución a la instrucción *clBuildProgram* (es decir, a la hora de crear el objeto que contiene el programa de OpenCL). Un ejemplo de opción de compilación es la opción *-cl-fast-relaxed-math*, mediante la cual se indica que las operaciones en coma flotante pueden violar el estándar IEEE 754.

No obstante, es decisión del *vendor* tenerlas en cuenta o no. Por ejemplo, en el caso de la implementación OpenCL de NVIDIA (para sus GPUs) sí que se tiene en cuenta este parámetro, pues en este caso el programa OpenCL se crea en tiempo de ejecución. Sin embargo, la implementación OpenCL de Altera ignora estos parámetros, ya que el

programa OpenCL en su caso es un binario que se crea previamente a la ejecución del programa. En su lugar, Altera especifica que se puede pasar *-fp-relaxed* como parámetro al AOC a la hora de crear el binario.

Por la experiencia durante este proyecto usando este tipo de opciones, no ha sido posible verificar que éstas tienen un impacto en el rendimiento. Es posible que en los experimentos realizados haya quedado oculta una posible mejora por otros factores que tienen mayor impacto en el tiempo de ejecución de la aplicación.

6.2.10 *Barriers*

Las barriers son un punto de sincronización entre múltiples *work items*. Esta instrucción provoca una espera hasta que todos los *work items* afectados por el barrier hayan llegado a ejecutar esta instrucción y sus operaciones de memoria se hayan completado. Existen dos tipos de barriers en OpenCL:

- **Local**: El punto de sincronización es a nivel de work group.
- **Global**: El punto de sincronización es a nivel global, de todos los *work items*.

Siempre que se pueda, es mejor evitar los barriers, ya que se desperdicia tiempo de cómputo por la espera provocada por la sincronización. Prescindir de los barriers de la implementación original de *reduce* implica modificar el patrón de acceso a memoria y la forma de reparto de trabajo entre los *work items*, por lo que el impacto de evitar los barriers se cuantifica en conjunto con las relacionadas con la memoria.

6.2.11 *Half float*

Existen varios grados de precisión para números en coma flotante que cumplen el formato estándar IEEE 754. Los más habituales son:

- **Precisión doble**: ocupan 64 bits, de los cuales 1 indica el signo, 11 el exponente y 52 la mantisa.
- **Precisión simple**: ocupan 32 bits, de los cuales 1 indica el signo, 8 el exponente y 23 la mantisa.

Aunque no está extendido en implementaciones de los estándares C99 ó C++11, la API de OpenCL trae consigo el manejo de números *half float*, que tienen aún menos precisión y ocupan únicamente 16 bits. Esto puede producir mejoras en la transferencia de datos a memoria global (se envían menos bytes) y también en el uso de recursos de determinados dispositivos (como FPGAs). Sin embargo, la API de OpenCL limita en gran medida el uso que se puede hacer de los *half float*. Ésta es la razón por la que no ha sido posible implementar esta optimización.

6.2.12 *Representación en coma fija*

Las operaciones en coma flotante requieren del uso de recursos adicionales que normalmente empeoran el consumo de energía y la latencia del *pipeline* respecto a las operaciones en coma fija, tal y como justifica Xilinx [6], una de las empresas más importantes y reconocidas en el campo de las FPGAs. En términos de representación, un número en coma fija siempre tendrá un número determinado de bits que contienen el número de la parte entera y otro el de la parte decimal. La ventaja de esta representación es que las operaciones aritméticas funcionan de la misma forma que con números enteros, por lo que se hace uso de las unidades aritmético-lógicas destinadas a números enteros, que son más "baratas".

Las funciones mostradas a continuación muestran las funciones usadas para realizar conversiones a coma fija y coma flotante dependiendo del tipo de dato:

```
//x is the source data, q is the number of decimal bits
#define INT2FIXED(x, q) ( x << q )
#define FIXED2INT(x, q) ( x >> q )
#define FLOAT2FIXED(x, q) ( (long) (x * (1L << q)) )
#define FIXED2FLOAT(x, q) ( ((float) x) / (1L << q) )
```

Con el objetivo de comparar ambas formas de realizar aritmética en la FPGA, se ha llevado a cabo una implementación en coma fija a partir de la versión original en coma flotante. Cabe destacar que el tiempo de CPU se incrementa notablemente a causa del *overhead* que introducen las conversiones entre coma flotante y coma fija. Además, incluso dentro del kernel ha sido necesario hacer uso de coma flotante en algunos casos, con el objetivo de no perder precisión en los cálculos, tal y como se explica en el Apéndice A.8. Esto implica a su vez mayor tiempo de ejecución del kernel. El tiempo de CPU en el *host* que antes era despreciable, en las versiones desarrolladas con coma fija supone un 50 % del tiempo de IO.

6.2.13 *Múltiples binarios AOCX*

En caso de que múltiples kernels no quepan en un mismo binario, es posible crear múltiples binarios y cargarlos alternativamente durante el ciclo de vida de la aplicación, dependiendo del kernel que se quiera ejecutar. Esto es completamente transparente al programador, dado que es el *runtime* de Altera el que se encarga de reprogramar la FPGA cuando se quieran ejecutar los kernels. Es decir, cuando se hace uso de las llamadas *clEnqueueNDRangeKernel* o *clEnqueueTask*, comprueba cuál es el programa que está cargado en la FPGA y, en caso de que no sea el que corresponde al kernel lanzado, reprograma la FPGA con el programa correspondiente a este kernel. No obstante, este tiempo de reprogramación puede ser tan grande que llegue

a ocultar el resto del tiempo de la aplicación. De hecho, en las pruebas realizadas este tiempo de reprogramación es más de 3500 veces mayor que el tiempo de cálculo del propio kernel, por lo que en este caso no merece la pena usar varios programas OpenCL.

6.3 RESULTADOS

Para el kernel de *reduce* en la FPGA se han obtenido los resultados mostrados en la Tabla 4, que se pueden comparar en la Figura 13 (para facilitar la comparación se han omitido las versiones #10, #13 y #14). Para todas las versiones mostradas, excepto la original, se ha aplicado la optimización de alinear la memoria aparte de la mostrada en el nombre de la versión. La versión #1 incluye esta optimización, además de los parámetros *attribute*. Por lo tanto, se deduce que, en comparación con la versión original, la pequeña variación en el tiempo de ejecución del kernel es debido a los parámetros *attribute*. El único parámetro que puede afectar al rendimiento del kernel es el que indica al compilador que use operaciones vectoriales de 4 elementos, pero se desconoce la razón por la que esto afecta de forma ligeramente negativa al tiempo de ejecución. De la misma manera se deduce que el *speedup* de 19,79 en el tiempo de IO + CPU sobre la versión original es debida al alineamiento de la memoria en el *host*.

Tabla 4: Resumen del impacto en el tiempo de ejecución (ms) de todas las optimizaciones en la FPGA

Versión	Only kernel	I/O + CPU	Total
Original	98,7	5423,2	5521,9
(#1) Float + Attributes	115,4	274	389,4
(#2) Float + 1 work item	700,3	252,2	952,5
(#3) Float + Memory pattern A + 200 *work items*	317,6	296,4	614
(#4) Float + Memory pattern B + 200 *work items*	169	275	444,1
(#5) Float + Memory pattern B + 200 *work items* + Local memory + Attributes	490,9	273,9	764,9
(#6) Float + Memory pattern B + 200 *work items* + 3 kernels + Attributes	81,6	257,7	339,4
(#7) Float + Memory pattern B + 3 kernels + 200/100/50 *work items* (3-level pyramid) + Attributes	81,5	253,3	334,8
(#8) Float + Memory pattern B + 3 kernels + 150/75/30 *work items* (3-level pyramid) + Attributes	81	252,9	333,9
(#9) Float + Memory pattern B + 3 kernels + 75/50/25 *work items* (3-level pyramid) + Attributes	101,7	254,2	355,9
(#10) Fixed point + 1 work item + Attributes	5693,6	1196,2 [3]	6889,8
(#11) Fixed point + Memory pattern B + 600 *work items* + Attributes	301,2	547,2 [3]	848,4
(#12) Fixed point + Memory pattern B + 200 *work items* + 3 kernels + Attributes	104,2	465,6 [3]	569,9
(#13) Half float	NA[1]	NA[1]	NA[1]
(#14) *track* + *reduce* (two AOCX binaries)	655629 [2]	125288,4	780917,4

[1] no ha sido posible su implementación debido al limitado soporte de la actual API de OpenCL

[2] 327801,6 (*track*) + 327827,4 (*reduce*), esto contiene también el tiempo de reprogramar la FPGA

[3] el 33 % de este tiempo corresponde al *overhead* de las conversiones entre coma flotante y coma fija

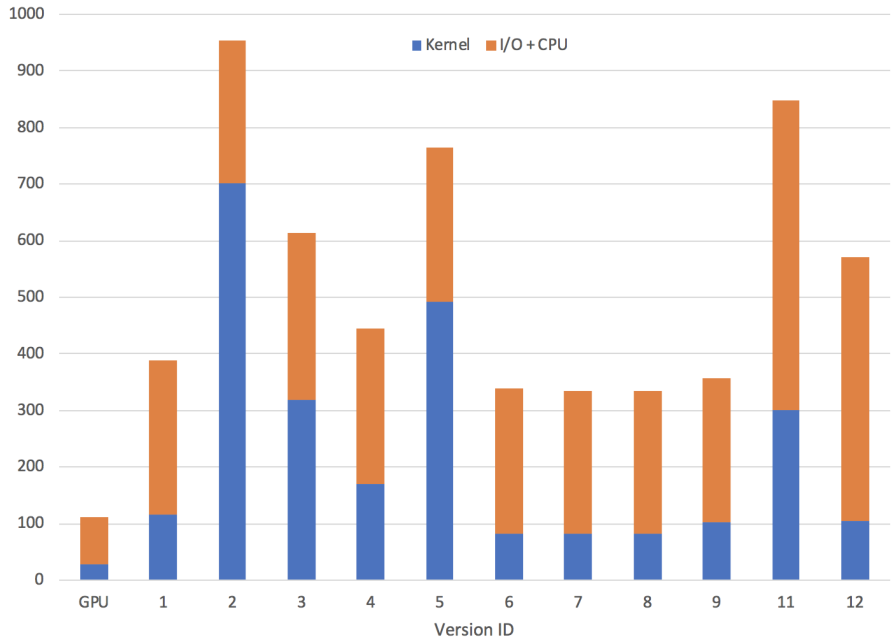


Figura 13: Tiempo (ms) de kernels + IO + CPU para cada una de las versiones con optimizaciones, de la 1 a la 12 corresponden a versiones para la FPGA

La versión #2 consiste en la ejecución de la implementación original pero con un único *work item*. Este es el mejor caso en términos de tiempo en IO + CPU, ya que la cantidad de datos leídos del buffer de salida del dispositivo depende directamente del número de *work items*. Sin embargo, el tiempo de ejecución del kernel es más de 7 veces mayor, ya que es un único *work item* quien tiene que hacer toda la reducción.

La versión #3 incluye una modificación sobre la manera de repartir el trabajo entre los *work items* y en consecuencia sobre el patrón de acceso a memoria. En esta versión, el trabajo asignado a cada *work item* es una zona de memoria contigua, por lo que siempre accede a posiciones de memoria consecutivas. Esto significa que el iterador privado de cada *work item* aumentará de uno en uno. Además, esta versión se ejecuta con 200 *work items*, razón por la que el tiempo de ejecución del kernel es menor que para la anterior versión, ya que el trabajo se reparte entre ellos. No obstante, este tiempo es 3,22 veces mayor que el de la versión original, lo que lleva a probar el patrón de memoria de la siguiente versión.

Para la versión #4 se ha modificado el patrón de memoria de manera que el iterador privado de cada *work item* aumenta cada iteración en tantas unidades como número de *work items* haya. Esto significa que la zona de memoria sobre la que trabaja cada uno ya no es contigua. Sin embargo, cada vez que la memoria entrega un bloque a un *work item*, los siguientes *work items* pueden aprovechar el resto de datos de este bloque, ya que corresponden a las direcciones de memoria

de su trozo de trabajo. Esta modificación supone un *speedup* de 1,88 sobre el tiempo de ejecución del kernel respecto a la anterior versión, aunque es 1,71 veces mayor que el de la versión original.

En la versión #5 se ha implementado además el uso de memoria local aunque, como se puede observar, esta implementación provoca un tiempo de ejecución del kernel 2,9 veces mayor que la anterior versión. Aunque no se ha hecho un análisis más exhaustivo, es probable que la causa de este aumento sea que la reducción final sobre los resultados parciales de cada *work group* es realizada por un único *work item* perteneciente a ese *work group* en vez de por varios (que podrían aprovechar todo el bloque leído de memoria local).

En la versión #6 con respecto a la versión #4 se aplica la optimización de dividir en tres kernels con parámetros *attribute* para que el AOC tenga más información en tiempo de compilación. Como se ha mencionado previamente, esto ha provocado un incremento en el uso de los recursos a la vez que se ha reducido el tiempo de ejecución del kernel, siendo ésta la primera versión con mejor tiempo de ejecución y de IO + CPU que la original.

Se desconocen los motivos por los que el tiempo de IO + CPU difiere en gran medida en las versiones #3, #4, #5 y #6, pues la cantidad de datos que se envían es en todas las versiones la misma, mientras que lo único que cambia es el propio kernel que va a ejecutarse en el dispositivo.

Para la versión #7 se ha variado el número de *work items* para cada nivel de la pirámide de manera que para el nivel con tamaño de imagen más grande (320x240) se lanzan 200 *work items*, para el intermedio 100 y para el más pequeño 50. Variar el tamaño de imagen más grande tendrá más impacto que variar el nivel más pequeño, ya que tiene más *pixels* que procesar. Las versiones #8 y #9 únicamente varían en el número de *work items* que se lanzan para cada nivel. Como se ha explicado anteriormente, reducir el número de *work items* implica que el tamaño del buffer de salida del dispositivo que tiene que leer el *host* es menor, por lo que el tiempo de su lectura también se reduce. No obstante, el tamaño de los buffers de entrada es tan alto que el tiempo que se consume enviándolos termina ocultando el tiempo de lectura del buffer de salida. Es por ello que se han probado distintas versiones, intentando reducir el número de *work items* a la cantidad óptima para cada nivel de la pirámide, de manera que el tiempo de IO + CPU (que es lo que más consume en todas las versiones) sea el menor posible sin aumentar demasiado el tiempo de ejecución del kernel en sí.

Las versiones #10, #11 y #12 corresponden a la implementación en coma fija junto con optimizaciones que se han aplicado a las versiones anteriores en coma flotante. De estas versiones cabe destacar que un 33 % del tiempo de IO + CPU es causa del *overhead* que introducen las conversiones entre coma flotante y coma fija, además de las

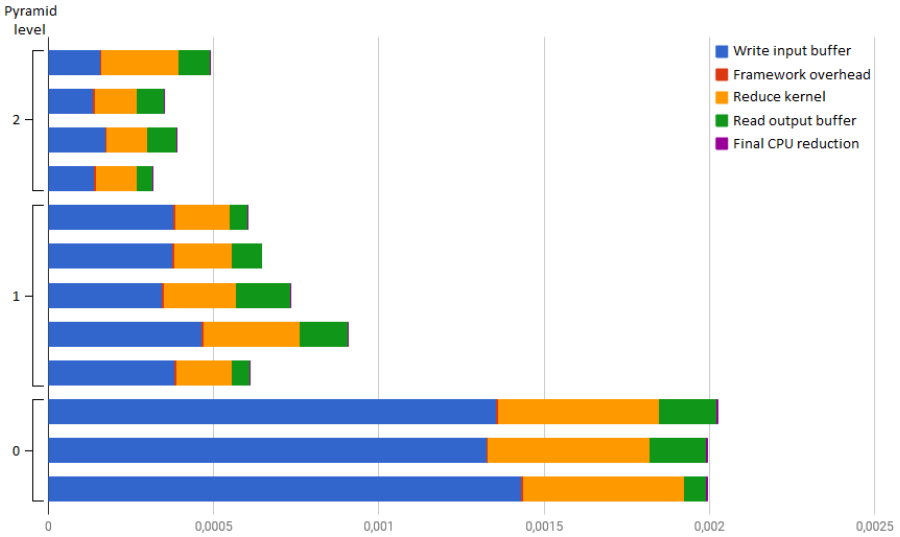


Figura 14: Distribución del tiempo (ms) consumido por la mejor versión de *reduce* en la FPGA de Socarrat

conversiones necesarias dentro del kernel para no perder precisión en los cálculos, tal y como se explica en el Apéndice A.8. Estos hechos hacen inviable una implementación en coma fija del kernel de *reduce*.

Por lo tanto, la mejor de las versiones es la #8, consiguiendo un *speedup* total de 16,54 sobre la versión original en la FPGA, aunque sigue siendo peor que la versión original en la GPU, concretamente 3 veces más lenta. En la Figura 14 se puede observar la distribución del tiempo que consumido por *reduce* en esta versión al procesar un *frame*. En el eje Y se distingue el nivel de la pirámide, siendo cada barra una iteración del bucle de convergencia dentro de ese nivel. En el eje X se muestra el tiempo consumido en milisegundos. Cabe destacar que la mayor parte del tiempo consumido es debido al envío del buffer de entrada, siendo esto más notable en el nivel con el tamaño de imagen más grande. El tiempo de lectura de los datos es en todos los casos bastante parecido, debido a que la esta penalización es causada principalmente por la alta latencia en la comunicación entre *host* y *device* más que por la cantidad de datos transferidos.

En conclusión, las optimizaciones sobre el kernel de *reduce* que han demostrado mejorar en alguna medida el rendimiento de la aplicación en la FPGA son las siguientes:

- Uso de memoria alineada para la escritura y lectura de buffers del device.
- Cambiar el patrón de acceso a memoria para que todos estos accesos se hagan de manera secuencial.
- División en tres kernels (uno por cada nivel de la pirámide) con parámetros de tamaño conocidos en tiempo de compilación.

- Diferente número de *work items* dependiendo del nivel de la pirámide.
- Uso de *REQD_WORK_GROUP_SIZE* para fijar la cantidad de *work items* que serán invocados para ejecutar el kernel (conocido en tiempo de compilación).

7 CONCLUSIONES Y TRABAJO FUTURO

Se ha evaluado el algoritmo KinectFusion en un sistema 3-heterogéneo compuesto por CPU, GPU y FPGA, poniendo especial foco en la FPGA. Dado que la FPGA tiene recursos limitados y no es posible implementar todo el algoritmo en ella, se ha identificado el kernel de *reduce* como el principal *hotspot* en el que este dispositivo podría mejorar el rendimiento de la aplicación. Se ha medido el impacto de diversas optimizaciones aplicadas sobre este kernel en el rendimiento y precisión de la aplicación. Además, se ha probado una implementación en coma fija aunque los resultados no han sido mejores que con coma flotante, en parte debido a que este kernel no es adecuado para ello por operar con datos con requerimientos de precisión entera y decimal muy dispares. Tras comparar los resultados de la implementación original y de todas las optimizaciones probadas, se concluye que el mayor inconveniente de la FPGA respecto a la GPU en esta plataforma concreta es la escritura o envío de buffers. A pesar de esto, se ha comprobado que con la FPGA es posible alcanzar rendimientos similares a la GPU.

Para complementar este proyecto se distinguen las siguientes posibles líneas de trabajo futuro:

- Investigar e implementar un método de balanceo de carga sobre el kernel de *reduce* que mejore el rendimiento total de la aplicación. En el Apéndice A.10 se encuentra un estudio previo sobre los posibles métodos de balanceo de carga que podrían funcionar.
- Uso de la CPU como otro dispositivo adicional dentro del entorno OpenCL (además de *host*).
- Investigar otros patrones de acceso a memoria para el kernel de *reduce* que puedan explotar mejor las capacidades de *pipelining* de la FPGA.
- Análisis y optimización del *pipeline* no sólo teniendo en cuenta los tiempos de ejecución, sino el consumo energético total de la plataforma.

Por último, en el Apéndice A.11 se encuentra un resumen de los esfuerzos totales invertidos en este proyecto.

APPENDIX

A.1 OPENCL SAMPLE APPLICATION

```
// let's suppose there's only a single OpenCL platform
cl_platform_id my_ocl_platform;
// let's suppose there's only a single device in our OpenCL
    platform
cl_device_id my_ocl_device;
// represents the compiled OpenCL code that will be run in the
    device
cl_program my_ocl_program;
// kernel, i.e. a function of the compiled code, whose purpose is
     adding to vectors
cl_kernel vector_add;
// represents the OpenCL context, i.e., relationship between host
     and list of devices belonging to the same OpenCL platform (e
    .g. host and two NVidia GPUs)
cl_context my_ocl_context;
// represents the queue between host and a single device in order
     to run tasks (pieces of code)
cl_command_queue my_ocl_command_queue;

// represents vectors A and B in device memory
cl_mem device_vect_a, device_vect_b;
// vectors of ints in host memory
const unsigned int vect_length = 4096;
cl_int host_vect_a[vect_length], host_vect_b[vect_length];

// used to check whether an OpenCL call has raised an error
cl_int retrieved_code;

int main(int argc, char **argv) {
        setup_ocl_env();
        setup_buffers();

        write_buffers();
        send_tasks();
        read_buffers();

        for(unsigned int i=0; i<vect_length; i++){
                printf("result[%d] = %d\n", i, vect_a[i]);
        }
}

void setup_ocl_env() {
        // does some OpenCL calls in order to setup everything
```

```
        // it's not relevant for this sample
}

void setup_buffers() {
        device_vect_a = clCreateBuffer(my_ocl_context,
            CL_MEM_READ_WRITE, vect_length * sizeof(cl_int), NULL
            , &retrieved_code);
        check_err(retrieved_code);

        device_vect_b = clCreateBuffer(my_ocl_context,
            CL_MEM_READ_WRITE, vect_length * sizeof(cl_int), NULL
            , &retrieved_code);
        check_err(retrieved_code);
}

void check_err(cl_int status_code) {
        // checks whether status_code is an error code
        // it's not relevant for this sample
}

void write_buffers() {
        // update buffers in the device
        retrieved_code = clEnqueueWriteBuffer(
            my_ocl_command_queue, device_vect_a, CL_TRUE, 0,
            vect_length * sizeof(cl_int), host_vect_a, 0, 0, 0);
        check_err(retrieved_code);

        retrieved_code = clEnqueueWriteBuffer(
            my_ocl_command_queue, device_vect_b, CL_TRUE, 0,
            vect_length * sizeof(cl_int), host_vect_b, 0, 0, 0);
        check_err(retrieved_code);
}

void send_tasks() {
        // Set the kernel arguments
        unsigned char arg = 0;
        retrieved_code = clSetKernelArg(vector_add, arg++, sizeof
            (cl_mem), (void *) &device_vect_a);
        check_err(retrieved_code);
        retrieved_code = clSetKernelArg(vector_add, arg++, sizeof
            (cl_mem), (void *) &device_vect_a);
        check_err(retrieved_code);

        // OpenCL needs three dimensions
        // for sake of simplicity only one is used
        size_t global_work[3] = { vect_length, 1, 1 };

        retrieved_code = clEnqueueNDRangeKernel(
            my_ocl_command_queue, vector_add, CL_TRUE, NULL,
            global_work, NULL, 0, 0, 0);
        check_err(retrieved_code);
}
```

```
void read_buffers() {
        // synchronous wait until all tasks are finished by the
            device
        retrieved_code = clFinish(my_ocl_command_queue);
        check_err(retrieved_code);

        // read the result into variable vect_a
        retrieved_code = clEnqueueReadBuffer(my_ocl_command_queue
            , device_vect_a, CL_TRUE, 0, vect_length * sizeof(
            cl_int), host_vect_a, 0, 0, 0);
        check_err(retrieved_code);
}

__kernel void kernel_vect_add(
                            // __global needed to specify this
                                memory address is globally
                                visible (i.e. also visible by
                                the host)
                            __global int * a,
                            __global int * b) {
        // Get my thread identifier for the first dimension,
            which represents vect_length
        uint threadId = get_global_id(0);
    // Compute my element
        a[threadId] += b[threadId];
}
```

A.2 FILTRO BILATERAL

Aplicar un filtro bilateral en una imagen 2D sirve para suavizar las superficies a la vez que se conservan los bordes/esquinas de los elementos que aparecen en ella. En la Figura 15 aparece en la parte izquierda la imagen de entrada y en la parte derecha el resultado obtenido tras aplicar este filtro. A grandes rasgos, un filtro bilateral modifica el valor de todos los píxeles de la imagen (en el ejemplo mostrado es el color) de tal manera que el nuevo valor es una media del valor de los píxeles adyacentes multiplicado por un peso. Este peso es generalmente obtenido de una distribución normal, dependiendo de la diferencia entre el valor del píxel adyacente y el píxel para el que se está calculando el nuevo valor. De esta forma se le da más influencia a los píxeles con valores más próximos (es decir, que tienen colores más parecidos).

A.3 INFORME GENERADO POR EL AOC

En la Figura 16 se observa una de las partes del informe que genera la herramienta AOC. En éste, se encuentra información sobre todos los bloques lógicos que atraviesa el kernel durante su ejecución. Además, es posible obtener información como la latencia que tendrán los

Figura 15: Filtro bilateral aplicado sobre los Minaretes y el lago Minarete en el desierto Ansel Adams del Bosque Nacional Inyo, Sierra Nevada, Condado de Madera, California - KinectFusion: Real-Time Dense Surface Mapping and Tracking [25]

accesos a memoria, si generan o no paradas en el *pipeline*, etc. Esto puede ser de gran utilidad para identificar cuellos de botella en la implementación del kernel. En la Figura 17), se aprecia con gran nivel de detalle la cantidad y tipo de recursos que utiliza cada instrucción del kernel.

A.4 USO DE RECURSOS DE LA FPGA EN LA IMPLEMENTACIÓN ORIGINAL

En la Tabla 5 se muestra la información que da el AOC sobre la estimación del uso de recursos del kernel de *reduce* en la FPGA de Socarrat para la implementación original. *Logic utilization* indica la cantidad de recursos totales que el kernel va a usar. Las ALUTs (*Adaptive Look-Up Tables*) son tablas que dada una entrada genera una salida determinada, configurable en tiempo de compilación del kernel por el AOC. Los *logic registers* son *Flip-flops*, la unidad más básica en lógica secuencial capaz de almacenar resultados entre ciclos. Los *memory blocks* son bloques de memoria más complejos, que pueden almacenar hasta 20000 *bits*. Estas memorias tienen algunos mecanismos que también están en las memorias RAM, como *bit* de paridad, detección y corrección de errores ECC, etc. Por último, los DSPs (*Digital Signal Processors*) son bloques diseñados para realizar operaciones en coma flotante. En la Tabla 6 se muestra la estimación de uso de estos recursos, excepto ALUTs (ya que la versión del compilador es más antigua y no da esta información), para la FPGA de DE1SoC. Como se puede

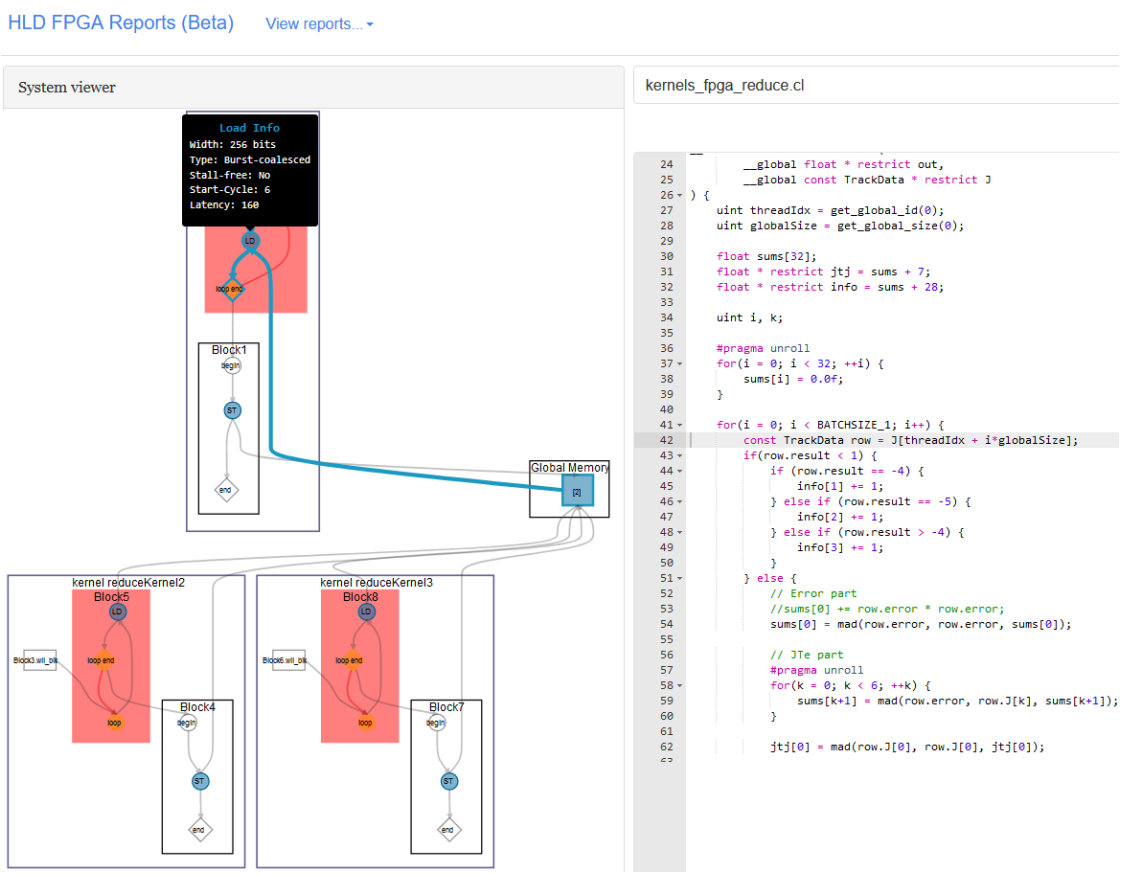


Figura 16: Informe de AOC con una vista del diseño creado por la FPGA

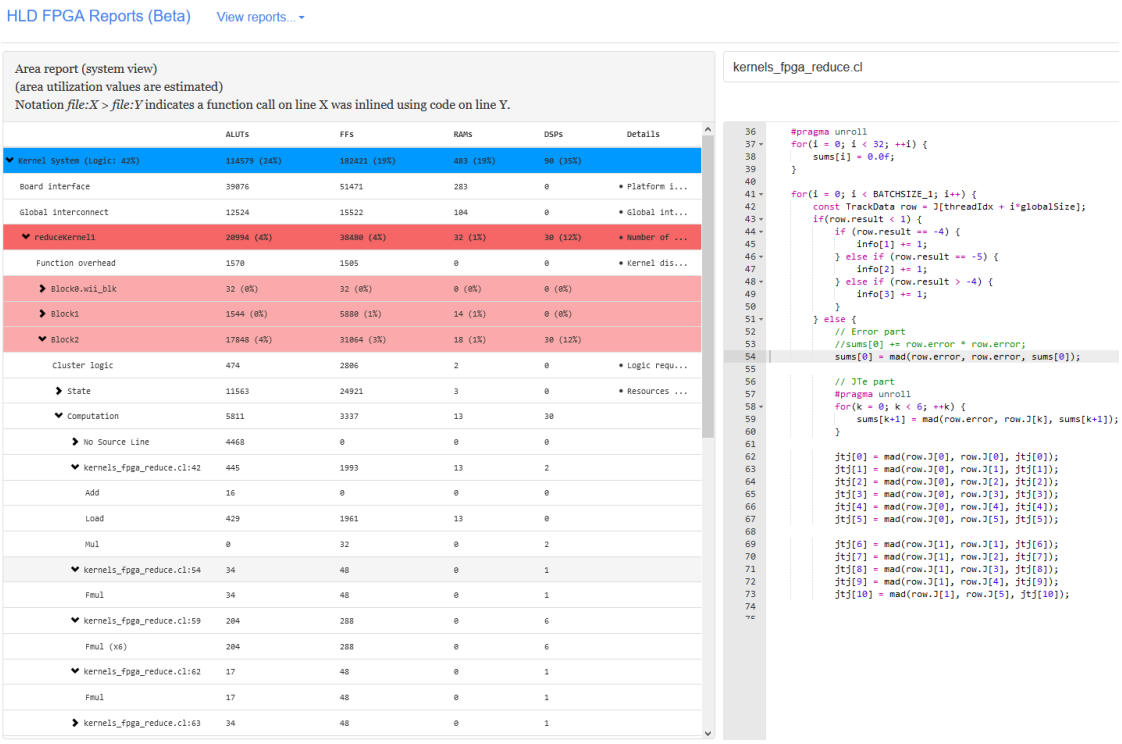


Figura 17: Informe de AOC con información detallada sobre cantidad y tipo de recursos utilizados por cada instrucción del kernel

observar, la utilización de recursos suele ser mayor en esta FPGA, ya que la cantidad de recursos es mucho más limitada.

En ocasiones, la ocupación de recursos real es mayor que la estimada por el AOC. En caso de que supere el 100 de los recursos, es necesario realizar modificaciones en el kernel que disminuyan este uso de recursos para que "quepa" en la FPGA.

Tabla 5: Porcentaje de uso de recursos en la FPGA de Socarrat de la versión original

Etapa	Kernel	Logic utilization	ALUTs	Logic registers	Memory blocks	DSP blocks
prepro	mm2meters	17	10	8	15	4
	bilateralFilter	25	15	11	23	11
track	halfSample	20	12	9	16	4
	depth2vertex	18	11	8	15	5
	vertex2normal	29	17	13	26	11
	track	31	19	14	24	25
	reduce	33	18	16	15	12
integrate	integrate	28	17	13	20	30
raycast	raycast	78	43	37	55	91
rendering	renderDepth	29	16	14	30	3
	renderTrack	31	19	14	24	25
	renderVolume	107	63	48	96	95

Tabla 6: Porcentaje de uso de recursos en la FPGA de DE1SoC de la versión original

Etapa	Kernel	Logic utilization	Logic registers	Memory blocks	DSP blocks
prepro	mm2meters	18	8	11	10
	bilateralFilter	50	23	45	25
track	halfSample	30	14	16	10
	depth2vertex	25	11	11	12
	vertex2normal	61	26	58	24
	track	77	33	55	57
	reduce	96	53	45	27
integrate	integrate	62	28	39	69
raycast	raycast	311	150	248	176
rendering	renderDepth	62	31	88	6
	renderTrack	48	24	61	2
	renderVolume	425	192	447	190

A.5 KERNEL ORIGINAL DE REDUCE

Como su propio nombre indica, este kernel realiza una reducción, con lo que recibe una cantidad de datos de entrada relativamente grande con respecto al tamaño de los datos de salida. Este trabajo es repartido entre 8 *work groups*, que a su vez se reparte entre sus 64 *work items* (originalmente estas eran las cantidades óptimas para ejecutar el kernel en la GPU). Se lanza la ejecución del kernel, que se estructura de la siguiente forma:

1. El *work item* itera sobre todos los datos que corresponden a su porción de trabajo y realiza una reducción sobre ellos.
2. Deja su resultado en la memoria local del *work group*.
3. Espera a que todos los *work items* de su *work group* hayan terminado de hacer cada uno su trabajo parcial.
4. Si es uno de los primeros 32 *work items*, entonces hace una segunda reducción parcial sobre el resultado de la anterior y deja el resultado en la memoria global.

Finalmente, el *host* coge estos 32 resultados de esta última reducción parcial y hace la reducción final, que es el resultado al trabajo global.

Por lo tanto, durante toda esta etapa de *reduce* se han realizado tres niveles de reducción.

```
/*
 * File: kernels.cpp
 * Description: host code
 */

//omitting source code previous to running reduce

size_t RglobalWorksize[1] = { 64 * 8 };
size_t RlocalWorksize[1] = { 64 };

//launch kernel in the device
clEnqueueNDRangeKernel(cmd_queue, reduce_ocl_kernel, 1, NULL,
    RglobalWorksize, RlocalWorksize, 0, NULL, NULL);

//read partial results
clEnqueueReadBuffer(cmd_queue, ocl_reduce_output_buffer, CL_TRUE,
     0, 32 * number_of_groups * sizeof(float), reduceOutputBuffer
    , 0, NULL, NULL);

//final reduction using TooN (maths library)
TooN::Matrix<TooN::Dynamic, TooN::Dynamic, float, TooN::Reference
    ::RowMajor> values(reduceOutputBuffer, 8, 32);
for (int j = 1; j < number_of_groups; ++j) {
        //result for the global problem is delivered to values[0]
        values[0] += values[j];
}

//omitting source code after running reduce


/*
 * File: kernels.cl
 * Description: kernel implementation run in the device
 */
__kernel void reduceKernel (
                __global float * restrict out,
                __global const TrackData * restrict J,
                const uint2 JSize,
                const uint2 size,
                __local float * S
) {
        uint blockIdx = get_group_id(0);
        uint blockDim = get_local_size(0);
        uint threadIdx = get_local_id(0);
        uint gridDim = get_num_groups(0);

        const uint sline = threadIdx;

        float sums[32];
        float * jtj = sums + 7;
```

```
float * info = sums + 28;

for(uint i = 0; i < 32; ++i)
sums[i] = 0.0f;

//first partial reduction
//fixed number of iterations, since size is the same for
    every thread
for(uint y = blockIdx; y < size.y; y += gridDim) {
        //fixed number of iterations, since size is the
            same for every thread
        for(uint x = sline; x < size.x; x += blockDim ) {
                const TrackData row = J[x + y * JSize.x];
                //only condition which could cause
                    divergence for any item in this loop
                if(row.result < 1) {
                        info[1] += row.result == -4 ? 1 :
                             0;
                        info[2] += row.result == -5 ? 1 :
                             0;
                        info[3] += row.result > -4 ? 1 :
                            0;
                        continue;
                }

                sums[0] += row.error * row.error;

                for(int i = 0; i < 6; ++i)
                sums[i+1] += row.error * row.J[i];

                jtj[0] += row.J[0] * row.J[0];
                jtj[1] += row.J[0] * row.J[1];
                jtj[2] += row.J[0] * row.J[2];
                jtj[3] += row.J[0] * row.J[3];
                jtj[4] += row.J[0] * row.J[4];
                jtj[5] += row.J[0] * row.J[5];

                jtj[6] += row.J[1] * row.J[1];
                jtj[7] += row.J[1] * row.J[2];
                jtj[8] += row.J[1] * row.J[3];
                jtj[9] += row.J[1] * row.J[4];
                jtj[10] += row.J[1] * row.J[5];

                jtj[11] += row.J[2] * row.J[2];
                jtj[12] += row.J[2] * row.J[3];
                jtj[13] += row.J[2] * row.J[4];
                jtj[14] += row.J[2] * row.J[5];

                jtj[15] += row.J[3] * row.J[3];
                jtj[16] += row.J[3] * row.J[4];
                jtj[17] += row.J[3] * row.J[5];
```

```
                        jtj[18] += row.J[4] * row.J[4];
                        jtj[19] += row.J[4] * row.J[5];

                        jtj[20] += row.J[5] * row.J[5];

                        info[0] += 1;
                }
        }

        //deliver results to local memory
        for(int i = 0; i < 32; ++i)
        S[sline * 32 + i] = sums[i];

        //the thread will wait until all other threads of the
            work group reach this point
        barrier(CLK_LOCAL_MEM_FENCE);

        //divergence is determined by sline (thread ID) instead
            of input data (unknown at compilation time), leading
            to a known divergence pattern since only the first 32
             threads will run the if statement
        if(sline < 32) {
        //second partial reduction
                for(unsigned i = 1; i < blockDim; ++i)
                S[sline] += S[i * 32 + sline];
                out[sline+blockIdx*32] = S[sline];
        }
}
```

A.6 COMPARACIÓN DE ANCHOS DE BANDA DE LOS DISPOSITIVOS

En la Tabla 7 se muestran las medidas tomadas de tamaños de buffers y anchos de banda tras ejecutar SLAMBench con la mejor versión del kernel de *reduce* en la FPGA de Socarrat, una versión con algunas de las optimizaciones que tenían un uso de recursos aceptable en la DE1SoC y la versión original de *reduce* en la GPU.

El hecho de que el número de escrituras y lecturas entre *host* y *device* es distinto para todos los dispositivos es debido a que no todos disponen de los mismos recursos y por lo tanto la precisión de la aplicación varía ligeramente, lo que provoca diferencias a la hora de converger. Como en unos dispositivos se llega antes a la condición de convergencia, el número de veces que se ejecuta *reduce* es distinto y por lo tanto también lo es la cantidad de datos transferidos. Además, en todas las implementaciones se leen resultados de distinto tamaño tras ejecutar *reduce*. Esto se debe a que en las implementaciones para las FPGAs la cantidad de datos leídos varía en función del número de *work items* (con diferente valor en Socarrat que en DE1SoC) y en

la versión original ejecutada en la GPU esta cantidad es fija independientemente del número de *work items*.

El máximo ancho de banda alcanzado durante toda la ejecución de la aplicación corresponde al envío del buffer más grande, mientras que el ancho de banda total es calculado dividiendo la cantidad datos transferidos a lo largo de la aplicación entre el tiempo que ha costado transferirlos. La diferencia entre todos los tiempos es principalmente debida al *hardware*. La GPU dispone de un bus el doble de rápido que la FPGA de Socarrat. Además, es posible que el *runtime* de OpenCL de Altera no realice de una forma óptima la transferencia de datos entre *host* y dispositivo, dados los *logs* que se muestran en la implementación original sobre estas transferencias (mostrados en la Sección 6.2.5).

Tabla 7: Tamaños de buffers (MB), tiempos de IO (s) y anchos de banda (MB/s) del kernel de *reduce* para todos los dispositivos de la plataforma

	Socarrat	DE1SoC	GPU
Mínimo tamaño de buffer enviado	0,024	0,031	0,001
Mínimo ancho de banda alcanzado	110,4	26,2	54,2
Máximo tamaño de buffer enviado	2,343	2,343	2,343
Máximo ancho de banda alcanzado	2001,4	89,1	5387,9
Total de lecturas/escrituras	718	684	722
Total de MB transferidos	363,7	342,9	373,5
Total de tiempo de transferencia de datos	0,27	4	0,08
Ancho de banda total	1339,8	85,7	4693,8

A.7 FUNCIONES BUILT-IN

La API de OpenCL ofrece funciones divididas principalmente en los siguientes tipos:

- **Matemáticas o geométricas**. Por ejemplo *dot* para computar el producto escalar.
- Relacionadas con los **work-items**. Por ejemplo *get_global_id* para obtener el identificador único del thread.
- De sincronización, como *barrier* o *mem_fence*.
- Operaciones vectoriales como carga o almacenamiento de vectores en memoria, suma o resta de vectores, etc.
- Tratamiento de imágenes.

Además, la especificación de cada función indica los tipos de datos que debe soportar, que en la gran mayoría de los casos son los siguientes:

- **flotante** de baja precisión, simple precisión o alta precisión (tamaños 16, 32 ó 64 bits respectivamente).
- **vector** de 2, 4, 8 ó 16 **flotantes** de una de las precisiones anteriormente mencionadas.

Por ejemplo, según la especificación de OpenCL la función *fma* puede aceptar los siguientes esquemas:

- **fp{x} fma(fp{x}, fp{x}, fp{x})**, siendo x el tamaño en bits del número en coma flotante, que puede ser 16, 32 ó 64.
- **fp{x}[y] fma(fp{x}[y], fp{x}[y], fp{x}[y])**, es decir, vectores de y elementos $2, 4, 8 ó 16$ de tamaño x.

En la siguiente porción de código se muestran dos kernels que tienen un resultado equivalente en cuanto a los valores calculados. Sin embargo, es posible que el kernel que hace uso de funciones built-in sea más eficiente en términos de tiempo de cómputo o uso de recursos.

```
__kernel void myKernel (
        __global int * a,
        __global int * b,
        __global int * c
) {
        // Get my thread identifier
        uint threadId = get_global_id(0);
        a[threadId] = a[threadId]*b[threadId] + c[threadId];
}

__kernel void myOptimizedKernel (
        __global int * a,
        __global int * b,
        __global int * c
) {
        uint threadId = get_global_id(0);
        // Equivalent built-in function, which may be optimized
            by the vendor
        a[threadId] = fma(a[threadId], b[threadId], c[threadId]);
}
```

A.8 PÉRDIDA DE PRECISIÓN EN LAS VERSIONES DE COMA FIJA

En primer lugar, dado que los datos de las versiones en coma flotante ocupan 32 bits (tipo *float*), se decidió mantener el mismo tamaño de dato para las versiones en coma fija (en este caso *int*), con el fin de no aumentar el tiempo de IO, que ya era una gran parte del tiempo

total. Una vez escogido el tamaño de datos, es necesario determinar cuántos de estos 32 *bits* representarán la parte entera y cuántos la parte decimal de cada variable, dependiendo del rango de números en el que se muevan estas variables. Este rango de números para cada variable se ha determinado en base a la observación sobre sus valores durante la ejecución de los experimentos. En este caso, para la variable *error* (de la estructura *TrackDataFixedPoint*) se ha escogido representación en Q24, es decir, 24 bits para la parte decimal, mientras que para los valores de *J* se ha elegido Q8. En ocasiones es necesario hacer cálculos en los que ambas variables son necesarias, pero estos cálculos en coma fija no son posibles si no están representados en el mismo formato (el mismo Q). Este problema tiene dos posibles soluciones. La primera es convertir ambos al mismo formato sólo en el cálculo en el que ambas son requeridas, como por ejemplo a Q16. Esto implica una pérdida de precisión, que en el caso del kernel de *reduce* es tan excesiva que hace inviable esta solución. La otra es convertir ambas variables a coma flotante de nuevo, realizar los cálculos y convertir de nuevo el resultado a coma fija, lo que implica un *overhead* adicional en el tiempo de ejecución del kernel.

A.9 BALANCEO DE CARGA

Este apartado se centra en estudiar las estrategias de balanceo de carga que puedan mejorar los resultados de la aplicación en base a las medidas presentadas en apartados anteriores. Principalmente se pone el enfoque sobre las etapas de *track* y *reduce*, pues son dos de las que más tiempo consumen en el dispositivo más rápido de la plataforma, la GPU, además de que el kernel de *reduce* ha sido el principal objeto de investigación durante este proyecto.

La forma más simple de balanceo pasa por ejecutar cada kernel en aquel dispositivo que mejor se adecúe a esa carga. Aunque puede resultar lo más rápido en términos de desarrollo de la aplicación, esto no suele dar un resultado óptimo. Si aplicáramos esta estrategia al caso que nos ocupa, todas las etapas se lanzarían en la GPU, por lo que ni siquiera habría mejora respecto a la implementación original.

Otra estrategia sería balancear la carga entre ambos dispositivos de manera estática. Esto pasaría por comparar los tiempos de ejecución de los kernels en ambos dispositivos y dividir esta carga entre los dispositivos en función de estos tiempos. De esta forma, si en base a nuestros experimentos sabemos que la GPU es el doble de rápida que la FPGA en ejecutar kernel de *reduce* se podrían asignar 23 de la carga (en este caso la imagen a procesar) a la GPU y 13 a la FPGA. Por lo tanto, en el caso ideal se obtendría un *speedup* de 33 %.

De la misma manera, es posible hacer este reparto de la carga de manera dinámica. Es decir, se puede dividir la carga a procesar en diversos trozos de manera que se envían siempre al primer disposi-

tivo que se encuentre ocioso, hasta haber procesado todos los trozos. Esto normalmente introduce mayor *overhead* que la anterior estrategia debido al coste computacional de dividir la carga además de un incremento en el tiempo de IO al haber mayor comunicación con los dispositivos.

Otra de las estrategias pasa por solapar en la medida de lo posible la ejecución de ambas etapas, con el fin de que los dispositivos estén ociosos el menor tiempo posible. Para ello es necesario tener en cuenta las dependencias entre las diversas porciones de código que podríamos considerar solapar, así como el tiempo que tarda cada una de ellas en ejecutarse en los diferentes dispositivos. En el Apéndice A.10 se explica cómo podría ponerse en práctica esta estrategia balanceando la carga entre CPU y FPGA.

A.10 TIEMPO DESGLOSADO DE LAS ETAPAS DE *track* Y *reduce* EN LA FPGA

En el experimento que se va a utilizar a continuación se han compilado con AOC *track* y *reduce*, resultando en un único binario con ambos kernels, ya que la FPGA tiene recursos suficientes para implementar ambos. Por lo tanto, en este experimento todas las etapas se lanzan en la CPU excepto *track* y *reduce*, que son etapas adyacentes y se ejecutan ambas en la FPGA. Esto significa que, en cuanto a envío y recepción de datos, únicamente es necesario enviar del *host* al dispositivo los buffers de entrada de *track* y leer los buffers de salida de *reduce*, ya que el resultado de *track* quedará en la FPGA y podrá ser consumido por *reduce* sin pasar por el *host*.

En la Figura 18 se puede ver el procesado, únicamente de las etapas de *track* y *reduce*, de 6 frames. Con el fin de ver correctamente la gráfica, se ha juntado en el eje X el procesamiento correspondiente a todos los frames, aunque esto no corresponde a la realidad, ya que faltarían por representar en el gráfico el resto de las etapas (de antes y después). En este caso se ha dividido el código de ambas etapas en las siguientes porciones:

1. **writebufo_track**: Envío de buffers de input para el kernel de *track* antes de comenzar el bucle sobre la pirámide de tres niveles.

2. **writebuf1_track**: Envío de buffers de input para el kernel de *track* en cada iteración del bucle de la pirámide de tres niveles.

3. **convergenceLoop**: Bucle en el que se ejecutan los kernels de *track* y *reduce*, que incluye la lectura del buffer de output de *reduce*. No es posible dividir el este bucle en unidades más pequeñas que puedan solaparse, debido a las dependencias existentes tanto en el cuerpo de este bucle como entre iteraciones del mismo.

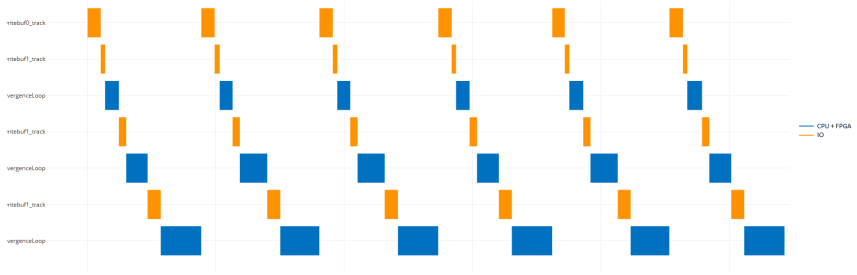


Figura 18: Tiempo desglosado de las etapas de *track* y *reduce* en la FPGA

En este caso el tiempo total de ambas etapas corresponde a la suma del tiempo en enviar una vez *writebufo_track* y tres veces *writebuf1_track* y *convergenceLoop* (uno por cada nivel de la pirámide). Sin embargo, se podría solapar la ejecución de los *convergenceLoop* de un nivel de la pirámide con la escritura de los buffers de entrada *writebuf1_track* del siguiente nivel. Es decir, mientras se ejecuta el *convergenceLoop* del primer nivel, el host podría a la vez enviar el buffer *writebuf1_track* del siguiente nivel, tal y como se muestra en la Figura 19 (representación sólo para un frame). De esta forma el tiempo total sería la suma del envío de *writebufo_track*, el envío del primer *writebuf1_track* y tres veces *convergenceLoop*. Incluso, ya que *writebuf1_track* no depende de *convergenceLoop*, podrían enviarse todos uno detrás de otro sin esperar al comienzo del siguiente nivel de la pirámide, resultando en la Figura 20. Por aclarar, estas dos últimas figuras no corresponden a experimentos reales, sino que son suposiciones sobre cómo se comportaría la aplicación en el caso de que se implementara el correspondiente balanceo de carga.

A.11 ESFUERZOS

En la Figura 21 se puede observar el tiempo invertido (en horas) en cada tarea a lo largo de este trabajo:

- **Configuración del proyecto y del entorno**. Esto comprende tareas como la instalación y configuración del *framework* OpenCL en el sistema, especialmente el compilador de OpenCL de Altera, necesario para trabajar con la FPGA. Dado que inicialmente no era posible usar SLAMBench en las plataformas Socarrat y DE1Soc, también fue necesario comprender el funcionamiento interno del *benchmark* para posteriormente realizar las modificaciones necesarias que lo hicieran funcionar. Esto incluye cambiar el proceso de automatización de *build* (compilación y ejecución), usando las herramientas *Make*, *CMake* y el *shell* de Linux, así como modificar el código C++ del propio *benchmark*.

- **Desarrollo e investigación**. Abarca tareas como investigación sobre el estado del arte de los sistemas heterogéneos y traba-

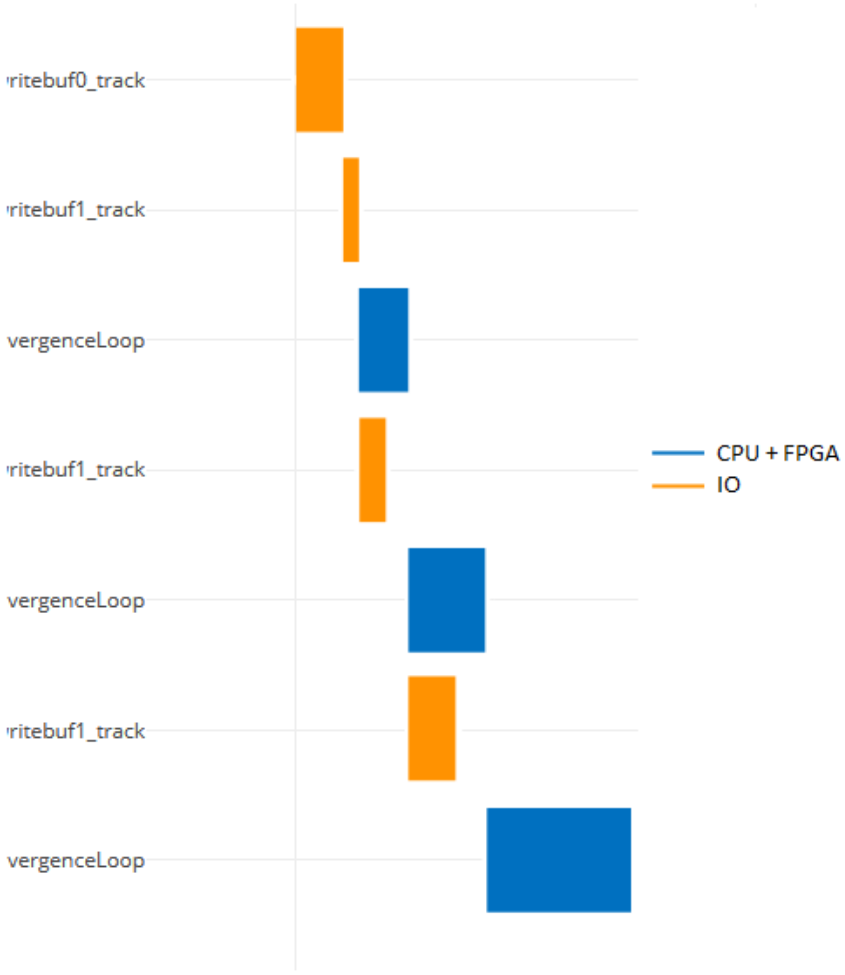


Figura 19: Tiempo desglosado de las etapas de *track* y *reduce* con solape entre *writebuf1_track* de un nivel de la pirámide y *convergenceLoop* del siguiente

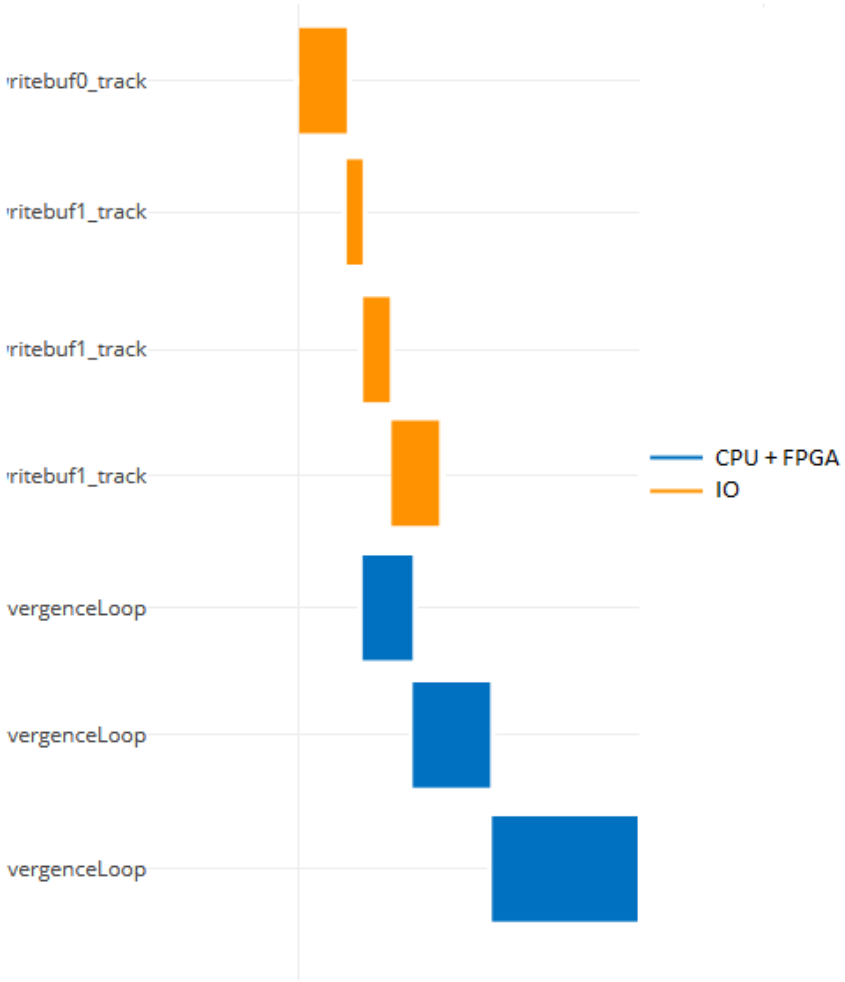


Figura 20: Tiempo desglosado de las etapas de *track* y *reduce* con solape entre *writebuf1_track* y *convergenceLoop*, independientemente del nivel de la pirámide

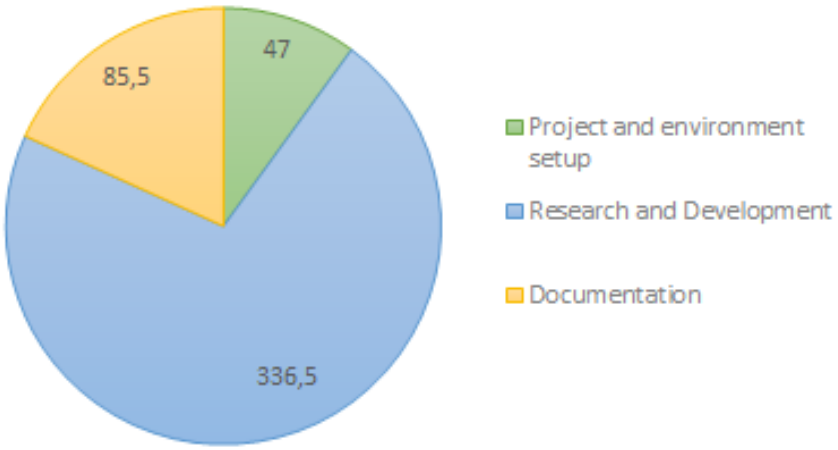


Figura 21: Esfuerzos invertidos (en horas) en cada tarea a lo largo de este trabajo

jos relacionados que puedan servir de ayuda para este proyecto, especialmente aquellos enfocados en el uso de FPGAs. Aparte, entender el funcionamiento de KFusion, analizar sus distintas etapas para estudiar los puntos críticos que tienen mayor impacto en el rendimiento de la aplicación e identificar aquellos que a priori son susceptibles de ser optimizados en una FPGA. Además, modificación del código C++ del propio *benchmark* para poder obtener métricas de grano más fino y *scripts* en Python para obtener gráficos y estadísticas sobre estas métricas para facilitar su análisis. Estos esfuerzos también abarcan el desarrollo de las técnicas exploradas de optimización, modificando la implementación en C++ y OpenCL de KFusion, así como la ejecución de los diversos experimentos y el análisis de sus resultados.

- **Documentación**. En general, todo lo relacionado a la preparación de este documento. Esto es síntesis de todo lo investigado y desarrollado a lo largo de este proyecto, así como búsqueda y elaboración de ilustraciones que faciliten su comprensión de cara al lector.

BIBLIOGRAFÍA

[1] Altera. *Altera SDK for OpenCL Best Practices Guide*. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_optimization_guide.pdf. [Online; páginas 1-61, 1-62; último acceso: 4 de Julio de 2018].

[2] Suleyman Demirsoy. *How OpenCL enables easy access to FPGA performance?* https://www.oerc.ox.ac.uk/sites/default/files/uploads/OpenCL_presentation_oxford.pdf. [Online; diapositiva 29; último acceso: 4 de Julio de 2018].

[3] Dmitry Denisenko. *OpenCL for FPGAs*. https://cpufpga.files.wordpress.com/2016/04/opencl_for_fpgas_isca_2016.pdf. [Online; diapositiva 35; último acceso: 4 de Julio de 2018].

[4] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous y Andre R LeBlanc. «Design of ion-implanted MOSFET's with very small physical dimensions». En: *IEEE Journal of Solid-State Circuits* 9.5 (1974), págs. 256-268.

[5] Andrew Ling Desh Singh Tom Czajkowski. *Tutorial: Harnessing the Power of FPGAs using Altera's OpenCL Compiler*. http://tcfpga.org/fpga2013/openCL_tutorial.pdf. [Online; último acceso: 4 de Julio de 2018].

[6] Ambrose Finnerty y Hervé Ratigner. *Reduce Power and Cost by Converting from Floating Point to Fixed Point*. https://www.xilinx.com/support/documentation/white_papers/wp491-floating-to-fixed-point.pdf. [Online; último acceso: 26 de Agosto de 2018].

[7] Khronos OpenCL Working Group. *The OpenCL Specification*. https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf. [Online; último acceso: 18 de Julio de 2018].

[8] Ankur Handa, Thomas Whelan, John McDonald y Andrew J Davison. «A benchmark for RGB-D visual odometry, 3D reconstruction and SLAM». En: *Robotics and automation (ICRA), 2014 IEEE international conference on*. IEEE. 2014, págs. 1524-1531.

[9] Sergio Iannace. «SLAMBench: evaluating OpenCL- based FPGA design for SLAM application». Tesis de mtría. Imperial College London, 2015.

[10] Intel. *Intel FPGA SDK for OpenCL Pro Edition Programming Guide*. https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf. [Online; página 9; último acceso: 19 de Julio de 2018].

[11] Qi Jia y Huiyang Zhou. «Tuning stencil codes in OpenCL for FPGAs». En: *Computer Design (ICCD), 2016 IEEE 34th International Conference on*. IEEE. 2016, págs. 249-256.

[12] David Patterson John Hennessy. *A New Golden Age for Computer Architecture, Domain-Specific Hardware/Software Co-Design, Enhanced Security, Open Instruction Sets, and Agile Chip Development*. https://www.youtube.com/watch?v=3LVeEjsn8Ts. [Online; último acceso: 16 de Julio de 2018].

[13] Lennart Johnsson. *The Impact of Moore's law and loss of Dennard Scaling*. https://indico.cern.ch/event/397113/contributions/1837780/attachments/1215934/1775678/Talk_2016-01-21.pdf. [Online; diapositiva 24; último acceso: 16 de Julio de 2018]. 2015.

[14] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers y col. «In-datacenter performance analysis of a tensor processing unit». En: *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE. 2017, págs. 1-12.

[15] Gordon E Moore. «Cramming more components onto integrated circuits». En: *Proceedings of the IEEE* 86.1 (1998), págs. 82-85.

[16] Fahad Bin Muslim, Liang Ma, Mehdi Roozmeh y Luciano Lavagno. «Efficient FPGA implementation of OpenCL high-performance computing applications via high-level synthesis». En: *IEEE Access* 5 (2017), págs. 2747-2762.

[17] Luigi Nardi, Bruno Bodin, M Zeeshan Zia, John Mawer, Andy Nisbet, Paul HJ Kelly, Andrew J Davison, Mikel Luján, Michael FP O'Boyle, Graham Riley y col. «Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM». En: *Robotics and Automation (ICRA), 2015 IEEE International Conference on*. IEEE. 2015, págs. 5783-5790.

[18] Richard A Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges y Andrew Fitzgibbon. «KinectFusion: Real-time dense surface mapping and tracking». En: *Mixed and augmented reality (ISMAR), 2011 10th IEEE international symposium on*. IEEE. 2011, págs. 127-136.

[19] Yuliang Pu, Jun Peng, Letian Huang y John Chen. «An efficient knn algorithm implemented on fpga based heterogeneous computing system using opencl». En: *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*. IEEE. 2015, págs. 167-170.

[20] *Repositorio GitHub con el código fuente*. https://github.com/mcanalesmayo/heterogeneous-slambench. [Online; último acceso: 5 de Septiembre de 2018].

[21] Kaz Sato. *An in-depth look at Google's first Tensor Processing Unit (TPU)*. https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu. [Online; último acceso: 18 de Julio de 2018].

[22] Qing Y Tang y Mohammed AS Khalid. «Acceleration of k-means algorithm using altera sdk for opencl». En: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 10.1 (2016), pág. 6.

[23] Anshuman Verma, Ahmed E Helal, Konstantinos Krommydas y Wu-Chun Feng. *Accelerating workloads on fpgas via opencl: A case study with opendwarfs*. Inf. téc. Department of Computer Science, Virginia Polytechnic Institute & State University, 2016.

[24] Hasitha Muthumala Waidyasooriya, Yasuhiro Takei, Shunsuke Tatsumi y Masanori Hariyama. «OpenCL-based FPGA-platform for stencil computation and its optimization methodology». En: *IEEE Transactions on Parallel and Distributed Systems* 28.5 (2017), págs. 1390-1402.

[25] U.S. National Park Service / Phrood (Wikipedia user). *Filtro bilateral aplicado sobre los Minaretes y el lago Minarete en el desierto Ansel Adams del Bosque Nacional Inyo, Sierra Nevada, Condado de Madera, California*. https://commons.wikimedia.org/wiki/File:Bilateral_Filter.jpg. [Online; último acceso: 12 de Julio de 2018].