Creación de un clúster de computación científica basado en FPGAs de bajo coste y consumo

Mariano Hernández García

Máster en Ingeniería Informática, Facultad de Informática.

Universidad Complutense de Madrid

Trabajo de Fin de Máster en Ingeniería Informática

Convocatoria: 5 julio 2017

Calificación obtenida: 10

Directores:

Guillermo Botella Juan

Alberto del Barrio García

Creación de un clúster de computación científica basado en FPGAs de bajo coste y consumo

Mariano Hernández García

Máster en Ingeniería Informática, Facultad de Informática.

Universidad Complutense de Madrid

Trabajo de Fin de Máster en Ingeniería Informática

Convocatoria: 5 julio 2017

Calificación obtenida: 10

Directores:

Guillermo Botella Juan

Alberto del Barrio García

Autorización de difusión

Mariano Hernández García

Madrid, a 5 de julio de 2017

El abajo firmante, matriculado en el Máster en Ingeniería Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Creación de un clúster de computación científica basado en FPGAs de bajo coste y consumo", realizado durante el curso académico 2016-2017 bajo la dirección de Guillermo Botella Juan y Alberto del Barrio en el Departamento de Arquitectura de Computadores y Automática (DACYA), y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional *E-Prints* Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en internet y garantizar su preservación y acceso a largo plazo.

Fdo: Mariano Hernández García

Agradecimientos

En primer lugar, quiero agradecer a mis dos directores, Alberto del Barrio y Guillermo Botella, el apoyo que me han dado a la hora de realizar este proyecto. Han sabido guiarme durante todo el proceso y, en algunos momentos difíciles, me han animado para que continuara. También me han ayudado a resolver algunos problemas y a redactar este documento correctamente. En segundo lugar, quiero agradecer a mi familia y amigos el apoyo que me han dado tanto en los buenos, como en los malos momentos. Sin ellos no habría llegado al lugar en el que estoy ahora.

Por último, quiero agradecer a la Facultad de Informática y a todo su personal, tanto docente como no, todo el esfuerzo que han realizado para que mi estancia allí haya sido la mejor posible y por permitirme adquirir de la mejor forma posible todos los conocimientos que he ido aprendiendo durante el período de mi Grado y Máster. Quiero agradecer de forma especial al grupo ArTeCS del Departamento de Arquitectura de Computadores y Automática (DACYA) el soporte que me han prestado, sobre todo a Francisco Daniel Igual Peña, y el acceso a las *Workstations* que he necesitado; así como agradecer las placas de desarrollo que me ha prestado el proyecto DESCARTES (Santander, UCM), referencia: PR26/16-20B-1.

Índice de contenidos

Índice de figuras

Índice de tablas

Resumen

En este trabajo se presenta la construcción de un clúster basado en FPGAs de bajo consumo energético y coste, capaz de ejecutar programas de alta complejidad, en el mismo o en menor tiempo que una estación de trabajo de mucho mayor coste y consumo. En la actualidad ya existen clústeres de este tipo, pero lo que diferencia al nuestro es que se han utilizado placas con FPGAs de bajas prestaciones y que se ha utilizado *OpenCL* como lenguaje de programación para acelerar la ejecución de los programas. Estas placas son las DE1-SOC de Altera y se caracterizan, aparte de por su bajo coste y consumo, por ser capaces de ejecutar un sistema operativo de base UNIX/Linux en su hard-core, un procesador ARM Cortex-A9 de dos núcleos. Sin embargo, las imágenes de UNIX/Linux disponibles tanto oficiales como no oficiales, presentan problemas de configuración o limitaciones. Debido a esto, se ha generado una imagen personalizada basada en *Debian* 8 y se ha instalado en ella el software necesario para poder ejecutar códigos escritos en *OpenCL* y compilados con el Kit de desarrollo de software de Intel para FPGAs. Se ha elegido esta distribución por ser muy utilizada, robusta y actualizada.

Además, se ha realizado una comparativa de los tiempos de ejecución, coste y consumo energético resultado de ejecutar un conjunto de 5 *benchmarks*, que hemos implementado en C y *OpenCL*, entre el clúster y una estación de trabajo o *Workstation* de altas prestaciones. Aunque en algunos casos los tiempos de ejecución de la *Workstation* han sido menores que los del clúster, el bajo consumo y coste de este último hace que su eficiencia energética sea mucho mejor que la de la Workstation y, por lo tanto, que sea una mejor opción.

Palabras clave: FPGA, clúster, C, OpenCL, UNIX/Linux, Benchmark, Workstation.

Abstract

This MSc Thesis presents the creation of a cluster based on low power FPGAs, capable of executing high complexity programs, in the same or in a shorter time than a Workstation of much greater cost and power consumption. Currently there are clusters of this type, but what set us apart is that we have used boards with low-end FPGAs in combination with OpenCL as a programming language to accelerate the execution of the programs. These boards are the Altera DE1-SOCs and besides their low power and cost they are characterized for being able to run a UNIX/Linux operating system on their hard- core, a dual core ARM Cortex A9 processor. However, the official and unofficial available UNIX/Linux images possess configuration problems or limitations. Because of this, a customized image based on Debian 8 has been generated and the necessary software has been installed on it to run codes written in OpenCL and compiled with the Intel Software Development Kit for FPGAs. This UNIX/Linux distribution has been chosen because it is very used, robust and updated.

Furthermore, we have compared the execution times, power consumption and costs between the cluster and a Workstation, by running 5 benchmarks that we have implemented in C and OpenCL. Although in some cases the Workstation's execution times have been lower than the cluster's, the low power consumption and cost of the cluster makes it more efficient and, therefore, a better option.

Keywords: FPGA, cluster, C, OpenCL, UNIX/Linux, Benchmark, Workstation

1. Capítulo I: Introducción

1.1 Motivación

Las grandes cantidades de datos que se generan a diario y la necesidad de procesarlas han aumentado la demanda de la capacidad de cómputo de nuestros sistemas informáticos. En un principio esta demanda se satisfacía elevando la potencia bruta de cálculo: procesadores con mayor número, más rápidos y eficientes núcleos, memorias más rápidas, tarjetas gráficas con mayores unidades de procesamiento, lenguajes de programación o incluso paradigmas completos más eficientes, uso de clústeres de computación, etc. Sin embargo, en los últimos años, y cada día más, el consumo de energía es cada vez más importante. No sólo necesitamos sistemas informáticos capaces de satisfacer las necesidades de procesamiento y almacenamiento actuales, sino que necesitamos urgentemente que lo hagan con el menor consumo energético posible.

Una de las soluciones más importantes en los últimos años ha sido el agrupamiento o *clustering* de sistemas informáticos, conectándose entre sí por medio de una red de altas prestaciones, con el fin de que se comporten como una única computadora. Estos clústeres hacen uso de la programación distribuida para ejecutar algoritmos en el menor tiempo posible (altas prestaciones o HPC) o para proporcionar un servicio continuo capaz de recuperarse ante los fallos (alta disponibilidad o HTC). Sin embargo, hemos llegado a un punto en el que cada vez cuesta más aumentar el rendimiento manteniendo un consumo energético moderado. Esto ha sido debido a que estamos alcanzando los límites del proceso de miniaturización de los transistores [1]. Es aquí donde se hace interesante la idea de utilizar computadoras más eficientes como nodos de estos clústeres. De forma general, podemos clasificarlos en 4 tipos: ASICs, FPGAs, ordenadores personales (PCs) y estaciones de trabajo (*Workstations)*. Los ASICs (*Aplication Specific Integrated Circuit*) son circuitos integrados creados específicamente para ejecutar una aplicación en particular. Son muy rápidos y eficientes energéticamente, pero son muy poco flexibles y su coste es muy elevado dado que se diseñan para una aplicación en concreto. De forma simplificada podemos definir a las FPGAs (Field Programmable Gate Array) como circuitos reprogramables que contienen bloques lógicos y conexiones, que pueden ser configurados las veces deseadas, por medio de un lenguaje de descripción especializado, para implementar un comportamiento o funcionalidad específicos. Se caracterizan por su alto rendimiento, menor tiempo de diseño y bajo consumo. Son capaces de superar el paradigma de la ejecución secuencial tradicional ya que aprovechan el paralelismo de *hardware.* Esto permite realizar un mayor número de operaciones por ciclo de reloj y, por lo tanto, ser muy eficientes. Al ser reprogramables, ofrecen flexibilidad y rapidez de prototipado. Se pueden diseñar sistemas complejos en ellas, depurarlos y mejorarlos en varias iteraciones. Gracias a estas características se acelera notablemente el desarrollo del producto final. Son mucho más eficientes energéticamente que cualquier computador personal o de uso científico. Esto, junto a su elevado rendimiento, hace que la relación coste/rendimiento sea mucho más elevada. Los ordenadores personales o PCs son computadores de propósito general, válidos para ejecutar aplicaciones de todo tipo, con una relación coste/rendimiento muy contenida. Sin embargo, están limitados para la realización de tareas de alta complejidad dadas sus limitaciones *hardware* en cuanto a capacidad de procesamiento, almacenamiento y comunicación. Por último, las estaciones de trabajo o *Workstations* son PCs con componentes *hardware* de altas prestaciones: múltiples *sockets* para instalar varias CPUs, CPUs de decenas de *cores* y varios *threads* por cada *core*, cantidades elevadas de memoria RAM, rápido y gran espacio de almacenamiento secundario y conexiones de red de alto rendimiento, como *Gigabit Ethernet* o *Infiniband*.

A medida que el sistema es más especializado el rendimiento y el coste de fabricación son mayores, mientras que el consumo energético es cada vez menor. Partiendo de esta premisa, los ASICs serían los dispositivos más rápidos y eficientes, pero más caros; irían seguidos de las FPGAs, los PCs y las *Workstations*. En la actualidad, la mayoría de clústeres están basados en el uso de ordenadores personales, servidores y estaciones de trabajo; que suelen tener instaladas una o varias tarjetas gráficas a modo de aceleradoras. Los ASICs se descartan por su alto coste de fabricación y su baja flexibilidad, pero las FPGAs reprogramables podrían ocupar un lugar diferenciador[1].

Aquí nace la idea y motivación del proyecto: el desarrollo de un clúster de FPGAs que, ejecutando algoritmos implementados en *C* y *OpenCL,* sea capaz de procesarlos en un tiempo aceptable, con un consumo muy inferior al de un ordenador personal o una estación de trabajo. Si bien esta idea no es nueva [2], lo que sí es novedoso es el empleo de *OpenCL* para la programación de los algoritmos ejecutados en el *clúster* con dichas FPGAs. En la actualidad podemos encontrar ejemplos reales que demuestran que el uso de *OpenCL* en FPGAs es un tema muy popular, por ejemplo, en temas relacionados con redes neuronales [3] o en el tratamiento de imágenes mediante convoluciones [4]. Como elemento nodo de este *clúster* hemos decidido utilizar la *Altera Cyclone V DE1-SOC,* dado que tienen un coste de $175 para universidades. Son apropiadas para este proyecto ya que son placas que cuentan con un procesador físico de doble núcleo, capaz de ejecutar un Sistema Operativo UNIX/Linux, y una *FPGA* reprogramable. Otra ventaja muy importante, y vital para este proyecto, es que permiten ser programadas en *OpenCL (Open Computing Language)* utilizando un kit de desarrollo *software* (SDK) creado por *Altera*.

1.2 Objetivos y plan de trabajo

El objetivo de este proyecto es demostrar que, con un conjunto de FPGAs, es posible construir un sistema de bajo coste y alta escalabilidad capaz de ejecutar algoritmos avanzados, en el mismo o menor tiempo que computadores de más coste; empleando para ello la programación basada en *C* y *OpenCL*.

Para conseguir este objetivo, el trabajo se ha dividido en las siguientes tareas:

- Realizar un estudio sobre las imágenes de Linux compatibles con la placa *Altera Cyclone V DE1-SOC* y determinar la más apropiada.
- Encontrar un conjunto de algoritmos representativos en *C* para realizar la comparativa. Traducirlos luego al lenguaje *OpenCL* y ejecutarlos en una placa.
- Crear el *clúster*, interconectando las placas entre sí y ejecutar dichos algoritmos en él de manera distribuida.
- Realizar una comparativa del rendimiento del *clúster* con una *Workstation* para determinar su viabilidad, en función del tiempo de ejecución, consumo, coste y escalabilidad.

[1] Son tan importantes que Intel ha comprado recientemente a Altera por unos $16,7 billones americanos: http://fortune.com/2015/08/27/why-intel-altera/

1.3 Motivation

The large amounts of daily generated data and the need to process them have increased the demand for computing capacity of our systems. At first, this demand was satisfied by raising the brute force calculation: processors with a larger number of cores and faster and efficient cores, faster memories, graphic cards with more computing cores, programming languages and programming paradigms oriented to increase the parallelism, the use of computing clusters ... However, in recent years power consumption has become the dominant factor while designing systems. We do not just require computing systems able of meeting the current processing and storage needs, but to meet a tight power budget too.

Nowadays, one of the most extended solutions consists of using clusters of computing elements connected to each other by high-performance networks to behave as a single system. These clusters use distributed programming to execute algorithms as fast as possible (High-Performance Computing, HPC) or provide continuous fault tolerant service (High-Throughput Computing, HTC). However, it is very hard to increase performance while keeping a low power consumption. One of the main reasons is that we are reaching the physical limit to scale transistors down [1]. Here comes the idea of using more efficient nodes as elements of the clusters. In general, we can classify them into 4 types: ASICs, FPGAs, personal computers and Workstations. The ASICs (Application Specific Integrated Circuit) are integrated circuits built for doing a specific application. They are very fast and efficient, but they are not flexible and very expensive to manufacture. In a simplified way, we can define the FPGAs (Field Programmable Gate Array) like reprogrammable circuits that contain logical blocks and connections that can be configured many times using a Hardware Description Language (HDL). They are characterized by their high performance, low design time and low power consumption. The FPGAs break with the paradigm of sequential programming because they allow to take advantage of parallelism at hardware level. This allow them to perform a greater number of operations per cycle and, therefore, increasing performance. As FPGAs can be reprogrammed, they offer flexibility and rapid prototyping features. Complex systems can be designed, debugged and improved in several iterations. This allows diminishing the time to market. This and their high performance makes them one of the most attractive solutions in terms of cost/performance tradeoff, better than the personal or even the powerful scientific computers. Personal computers or PCs are general purpose computers, valid to execute all kind of applications. Nevertheless, they are not optimized to run scientific programs that typically present a high parallelism degree. On the other hand, the Workstations are PCs with high performance hardware components: multiple sockets to install various CPUs, CPUs with many cores and many threads per core, large amounts of RAM memory, fast and large secondary storage and high performance network connections, such us Gigabit Ethernet or InfiniBand.

The more specific the system, the more efficient and expensive it is. On the other hand, the power consumption is usually lower. Thus, the ASICs are the fastest and most efficient hardware, but the most expensive as well. They are followed by the FPGAs, the PCs and the Workstations. Nowadays, the vast majority of computing clusters are based in personal computers, servers and Workstations. The ASICs are discarded because of their high manufacturing cost and low flexibility. But the FPGAs could play a differentiating role.

Here is where the idea and motivation of this project was born: creating a cluster for scientific programming with low-end FPGAs and low power consumption for running programs written in C or OpenCL faster and requiring less energy than a typical Workstation. This idea is not new [2], but what is novel is the use of OpenCL to program the algorithms executed on the cluster of these FPGAs. Nowadays, we can find real examples that show that the use of OpenCL

for programming FPGAs is a very popular trend, for example, in topics about neuronal networks [3] or image processing using convolutions [4]. The basic element of our computing cluster is the DE1-SOC. These boards perfectly suit the requirements of this project, because they are capable of running UNIX/Linux based Operating Systems on their hard core and because they have an integrated reprogrammable FPGA (Cyclone V by Altera). Another advantage that is very important for this project is that they can be programmed employing OpenCL by using the Inteľ s Software Development Kit (SDK) for FPGAs.

1.4 Goals and work plan

The main objective of this MSc Thesis is show that with a set of FPGAs it is possible to construct a low cost and highly scalabile system capable of executing advanced algorithms, in the same or less time than costly and power hungry computers of more costs.

To achieve this goal, the work plan consists of the following stages:

- Performing a study about the UNIX/Linux images compatible with the Altera Cyclone V DE1-SOC and determining the most appropriate to customize our image.
- Finding a set of C-based algorithms to create a benchmark suite in order to make a comparative. Afterwards, writing them in OpenCL and executing them in one DE1-SOC.
- Creating the cluster, connecting the boards together, and executing those algorithms on it in a distributed way.
- Comparing the performance of the cluster with a Workstation to determine its feasibility, based on execution time, power consumption, costs and scalability.

2. Capítulo II: Estado del Arte

En este apartado se ha realizado un estudio sobre las imágenes de UNIX/Linux que existen actualmente para las placas DE1-SOC, con el fin de determinar la más adecuada para lograr el objetivo del proyecto. Estas imágenes deben cumplir dos requisitos indispensables:

- Deben tener una conexión de red rápida, que pueda funcionar sin necesidad de introducir la contraseña a través del protocolo SSH. Esto es necesario para enviar y recibir la información a las placas de la forma más rápida posible y para evitar que, desde el nodo maestro, haya que escribir las credenciales de acceso para cada conexión.
- Deben tener instalado el *software* necesario para ejecutar aplicaciones escritas en *C* y *OpenCL* compiladas con la versión 16.0 del *Intel FPGA SDK for OpenCL*, que es la versión más reciente en la fecha que se empezó a escribir este Estado del arte y es la versión que se ha instalado en la estación de trabajo utilizada para compilar.

2.1 Instalación de una imagen de Linux en la DE1-SOC

Antes entrar en el análisis de cada una de las imágenes de Linux disponibles para la DE1-SOC, se va a ver el proceso necesario para ejecutarlas en las placas. Se seguirá el manual que se puede descargar desde la web de Altera [5]. Se necesitan los siguientes requisitos:

- Un PC con *Windows* o Linux.
- Una placa Altera *Cyclone* V DE1-SOC.
- Un cable micro USB a UART (viene incluido con la placa).
- Una tarjeta de memoria micro SD de, al menos 8GB de capacidad. Para la realización de este proyecto, se han utilizado micro SDs de la marca *Sandisk* UHS-I de 32GB.

El primer paso consiste en descargar y descomprimir una de las imágenes de Linux desde la web de Altera. Una vez hecho esto, se obtiene un archivo de extensión IMG. Este archivo tiene que ser copiado en una tarjeta micro SD y tiene que ser insertado en la ranura adecuada de la DE1-SOC. En *Windows* se puede utilizar el programa llamado *Win32 Disk Imager,* que puede ser descargado de forma gratuita. La Figura 1 muestra este proceso en Windows.

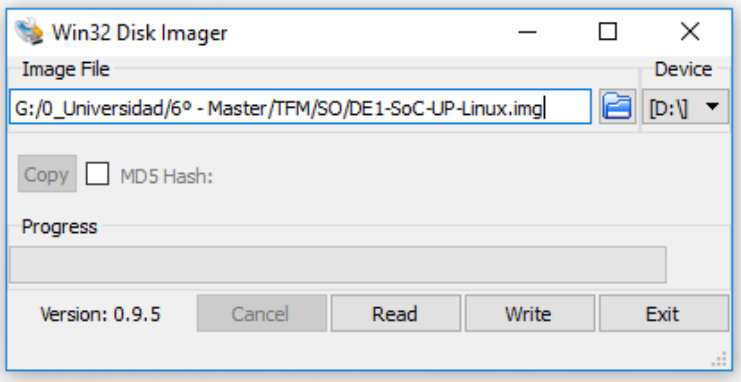


Figura 1: Captura del programa Win32 Disk Imager

En Linux se utiliza el comando *dd*[2]. Si suponemos que la micro SD es el dispositivo */dev/mmcblk0*, el comando a ejecutar sería el siguiente:

```
$ sudo dd if=/dev/mmcblk0 of=~/workspace/DE1-SoC-UP-Linux.img bs=1M
status=progress
```

Hay que configurar la DE1-SOC para que su procesador ARM *Cortex* A9 sea capaz de programar la FPGA incorporada en la misma. Para ello, la DE1-SOC cuenta con unos interruptores en su parte inferior cuyo valor puede ser '0' o '1'. A estos interruptores se les llama MSEL *switches.* Se configuran para que valgan 010101, de abajo a arriba, como ilustra la Figura 2.

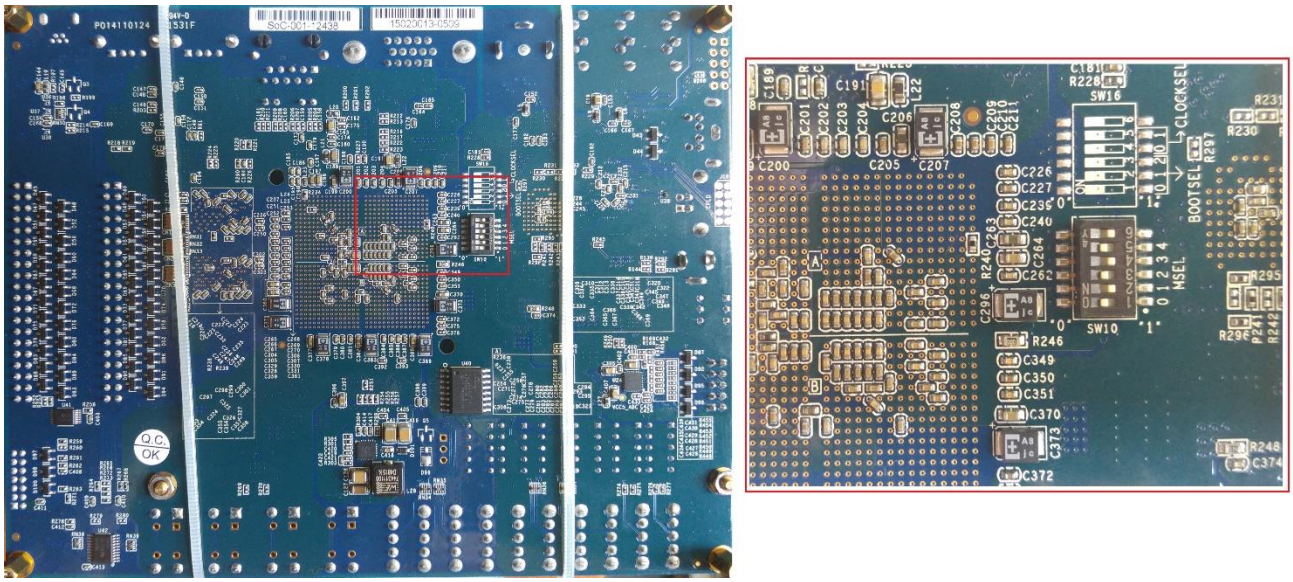


Figura 2: Configuración de los pines de la DE1-SOC para la instalación de una imagen de UNIX/Linux

Por último, se conecta la placa a un ordenador a través del puerto serie mediante el cable micro USB a UART (incluido con la DE1-SOC) y se utiliza un programa como *Putty* (*Windows*) o *screen* o *minicom* (Linux) configurado debidamente con las siguientes opciones:

- Baudios: 115200
- Data bits: 8
- Paridad: ninguna
- Control de flujo: ninguno

Se enciende la DE1-SOC pulsando sobre el interruptor de color rojo y en unos segundos se debería ver la secuencia de arranque con todos sus mensajes informativos. Ahora se puede acceder con las credenciales, configurar la red, instalar software, etc. A partir de este momento, lo recomendable sería conectarse por SSH a través del puerto Ethernet para enviar datos, cosa que no puede hacerse con la conexión por serie.

[2] https://en.wikipedia.org/wiki/Dd_(Unix)

2.2 Linux Angstrom

Se trata de una distribución de Linux basada en *Debian* creada especialmente para ser ejecutada en sistemas empotrados como las DE1-SOC [6] de este proyecto u otras placas, como la *BeagleBoard*. Es compatible con el proyecto *Yocto* [7], que es un proyecto de *software* libre que proporciona las plantillas, mecanismos y herramientas necesarias para crear distribuciones de UNIX/Linux para sistemas con arquitecturas no x86, como la ARM.

Aunque se puede compilar por nosotros mismos una distribución de Linux basada en el *kernel* de Angstrom, desde la web oficial de Altera se puede descargar una imagen preparada para ser instalada en la DE1-SOC [8]. Hay disponibles dos imágenes: *BSP for Altera SDK OpenCL 14.0* y *BSP for Altera SDK OpenCL 16.0*. Se decide probar la imagen de la versión 16.0, puesto que esa es la versión instalada del Kit de desarrollo de *software*, cuyas pruebas resumimos a continuación:

1. Se conecta a la DE1-SOC a través de la UART con el cable adaptador de UART a USB que viene con la placa y se arranca.

2. Ahora se prueba la conexión de red por SSH, pero se detectan dos problemas: la petición de la contraseña se demora bastantes segundos y, una vez solicitada, se obtiene un mensaje de error de autenticación que impide hacer *login* en la placa. El primer problema tiene que ver con una mala configuración del cliente DNS. Se modifica el fichero de configuración del servidor de SSH almacenado en */etc/ssh/sshd_config* y se descubre que los cambios realizados no surten efecto. Se instala otro servidor de SSH, pero tampoco parece funcionar. Pese a estar basada en *Debian*, la distribución utiliza su propio gestor de paquetes llamado *opkg* [9] y, debido al resultado de las pruebas realizadas, se infiere que la configuración utilizada no puede cambiarse. En cuanto al segundo problema, depurando y buscando una solución se descubre que se trata de un problema de cifrado que se soluciona añadiendo una opción extra al comando SSH. Ahora nos deja autenticarnos, pero la conexión resulta lenta y, por lo tanto, incumple uno de los requisitos que impusimos al principio.

3. Si tratamos de copiar un fichero con el comando SCP ocurre lo mismo: la conexión es lenta.

Pese a los problemas encontrados en la red, se intenta ejecutar los programas precompilados de *OpenCL* llamados *hello_world* y *vector_add*, pero se obtiene el siguiente error relacionado con la falta de una librería:

```
root@angsrom:~/OpenCL/helloWorld# ./hello_world
./hello_world: /lib/arm-linux-gnueabihf/libc.so.6: version
`GLIBC_2.17´ not found (required by ./hello_world)
```

Esta imagen no cumple con ninguno de los requisitos que hemos impuesto. Por un lado, la red presenta problemas de velocidad de conexión que no pueden solucionarse al tratarse de una distribución personalizada y muy limitada en cuanto a configuración, lo que podría ser un problema en un futuro de cara a la instalación de *software*. Por otro lado, se presenta un problema relacionado con falta de librerías que impide ejecutar un programa compilado con el *Intel FPGA SDK for OpenCL*. Por lo tanto, esta imagen queda descartada.

2.3 Linux BSP (Board Support Package)

Se tratan de cuatro imágenes de Linux que se pueden descargar también desde la web de Altera. Son las siguientes y tienen las siguientes características:

- ***Linux console**:* se trata de una distribución basada en el proyecto Angstrom como la anterior, pero, a diferencia de ella, viene sin el módulo necesario para ejecutar *OpenCL*.
- ***Linux console with framebuffer***: es igual que la anterior, solo que se ha utilizado la FPGA para que, a través del puerto VGA de la placa, se obtenga imagen. Esto posibilita conectar la placa a un monitor VGA y visualizar la terminal en él, pero tiene un gran inconveniente para nuestro proyecto: la FPGA queda inutilizada para ejecutar los algoritmos en *OpenCL.*
- ***Linux LXDE Desktop**:* como su nombre indica, se trata de una distribución basada en LXDE con interfaz gráfico a través del puerto VGA (de ahí su nombre *desktop*). Como en el caso anterior, la FPGA se utiliza para comunicar la señal de video a través del puerto VGA y, por lo tanto, queda inutilizada para ejecutar *OpenCL*. Se hacen las pruebas de red y se verifica que la red funciona sin problemas: la conexión es rápida y el intercambio de ficheros se hace a la máxima velocidad que permite la red, que es *Fast Ethernet*[3].
- ***Linux Ubuntu Desktop:*** igual que la anterior, pero basada en Ubuntu. La red también funciona sin problemas, pero la FPGA no puede utilizarse para acelerar los algoritmos. Queda descartada.

De todas ellas la más prometedora es la primera. Como hemos visto, viene sin el módulo necesario para cargar *OpenCL* pero, si la red funciona como debe, podríamos tratar de compilarlo e instalarlo por nuestra cuenta. Se hacen las pruebas pertinentes y se obtiene el mismo resultado que en el primer caso: la red es lenta y es imposible cambiar la configuración. Por lo tanto, descartamos instalar el módulo de *OpenCL*.

2.4 Imagen de Linux incluida en el Intel FPGA SDK for OpenCL

Se trata de la imagen de UNIX/Linux que viene incluida junto a la instalación del *software* de Intel necesario para realizar la compilación cruzada de los programas. En nuestro caso, se trata de la versión 16.0, que era la más actualizada en el momento de la realización de este estado del arte. Se comprueba que su base es *OpenBSD*.

Se repiten los pasos anteriores y se consigue auto configurar la red mediante el protocolo DHCP utilizando un *router*, pero la conexión por SSH no funciona. Pese a introducir correctamente las credenciales, se obtiene el mensaje de error *Connection refused* transcurridos unos 30 segundos. Nuevamente se debe a un error de DNA. Se modifica el fichero de configuración del servidor SSH y se reinicia la placa, pero los cambios no han surtido efecto. La configuración de la imagen se ha limitado y los cambios realizados no surgen efecto. Pese a ello, se ejecutan los programas precompilados *hello_world* y *vector_add* y se ejecutan con éxito. No

[3] https://en.wikipedia.org/wiki/Fast_Ethernet

obstante, al ser imposible hacer funcionar el servidor SSH, y por consiguiente el comando SCP para transferir datos por red, esta imagen queda nuevamente descartada.

2.5 Tabla comparativa

A modo de resumen, se han recogido los resultados anteriores en la Tabla 1, que se muestra a continuación:

Nombre	Sistema base	Kernel	Red	Funciona OpenCL versión 16.0
Linux Angstrom	Debian	Angstrom v3.13	Muy lenta (tanto al loguearse, como al intercambiar información)	Sí
Linux console	Debian	Angstrom v3.12	Muy lenta (tanto al loguearse, como al intercambiar información)	No, pero puede instalarse manualmente
Linux console with framebuffer	Debian	Angstrom v3.12	Muy lenta (tanto al loguearse, como al intercambiar información)	No y no puede instalarse
Linux LXDE Desktop	LXDE	3.12/4.5	Rápida (Fast Ethernet)	No y no puede instalarse
Linux Ubuntu Desktop	Ubuntu	3.12/4.5	Rápida (Fast ethernet)	No y no puede instalarse
Imagen incluida Intel FPGA SDK for OpenCL 16.0	OpenBSD	Angstrom v3.13	No funciona. El login falla y es lento	No. Error relacionado con la librería GLIBC_2.17

Tabla 1: Comparativa imágenes de Linux para la DE1-SOC

2.6 Conclusiones

Como hemos visto, ninguna de las imágenes cumple los 2 requisitos que se han impuesto al principio: que la red sea suficientemente rápida (de forma cualitativa) y que esté preparada para ejecutar los programas compilados con la versión 16.0 del SDK. Por ello se ha tomado la decisión de compilar por nuestra cuenta una imagen de UNIX/Linux e instalar en ella el módulo de *OpenCL*, versión 16.0. Pese a parecer una solución arriesgada, el hecho de generarla por nosotros mismos nos da el control absoluto sobre la configuración y el *software* instalado. Los detalles sobre la creación de esta imagen, la instalación del módulo de *OpenCL* versión 16.0 en ella se detalla, la creación del *clúster* y la realización de las pruebas, mediciones y comparativa con una Workstation se harán en el apartado siguiente.

3. Capítulo III: Propuesta

En este apartado vamos a ver cómo se ha creado la imagen de UNIX/Linux que se ejecutará en los nodos del *clúster*, cómo se ha instalado en ella lo necesario para ejecutar las aplicaciones compiladas con el *Intel FPGA SDK for OpenCL 16.0* y cómo instalarlo y configurarlo en cada una de las placas para construir el *clúster.*

3.1 Creación de una imagen de Linux propia e instalación de los *drivers* de *OpenCL*

En este apartado vamos a ver cómo hemos creado la imagen de Linux personalizada para la DE1-SOC. En primer lugar, vamos a ver los ficheros necesarios para ejecutar Linux, el particionado de la tarjeta de memoria y las etapas del proceso de arranque de la placa. Después veremos cómo generarlos y cómo compilar el resto de archivos necesarios. Por último, veremos cómo generar un archivo que permita instalar de una forma muy fácil nuestra imagen en cualquier placa DE1-SOC.

3.1.1 Ficheros necesarios y secuencia de arranque

Los ficheros necesarios para ejecutar un Sistema Operativo de base Linux en placas *Cyclone V*, como las DE1-SOC de este proyecto, son los siguientes:

- **socfpga.dtb:** fichero *Device Tree Blob.* Se trata de una estructura de datos que almacena la descripción del *hardware*.
- **soc_system.rbf:** archivo comprimido que contiene la configuración de la FPGA.
- **u-boot.src:** *script* que sirve para configurar la FPGA.
- **zImage:** fichero que contiene el *kernel* comprimido.
- **Sistema base:** ficheros y directorios de la distribución de UNIX/Linux*: Ubuntu, LXDE, Angstrom…* en nuestro caso, una *Debian 8*.
- **preloader-mkpimage.bin:** fichero que contiene las cabeceras necesarias por la etapa *BootROM*.
- **U-Boot image** [10]**:** Se trata de un proyecto de *software* libre para sistemas empotrados encargado de ser el cargador primario del arranque o *bootloader*.

Estos ficheros no pueden ser copiados sin más a una tarjeta micro SD, sino que tienen que ser copiados en unas particiones especificadas en el *Master Boot Record,* que contiene el tipo, la dirección de memoria de inicio y el tamaño de cada una de ellas [11]:

- **Partición 1:** se trata de una partición de tipo B, más conocida como FAT32, que contendrá el kernel del sistema (archivo zImage), los archivos relacionados con la configuración de la FPGA (*soc_system.rbf*) y la estructura de datos que almacena la descripción del *hardware (socfpga.dtb)*.

- **Partición 2:** se trata de una partición de tipo 83, más conocida como EXT, que contendrá los ficheros del sistema base. En nuestro caso, contiene los ficheros de la *Debian 8.*
- **Partición 3:** se trata de una partición de tipo A2 en la que los ficheros se almacenan en "crudo" (*RAW* en inglés). No tiene un nombre conocido como el de los casos anteriores. En esta partición se almacenan los ficheros *preloader-mkpimage.bin* y *U-Boot image* para que sean leídos por la etapa de *BootROM.*

Todo esto queda resumido en la Tabla 2.

Uso	Tipo	Ficheros
Espacio libre	-	-
Partición 3	A2 (RAW)	Preloader image U-boot image
Partición 2	83 (EXT Linux)	Sistema Base Linux (Debian)
Partición 1	B (FAT32 Windows)	*Socfpga.dtb* *Soc_system.rbf* *u-boot.src* *zImage*
U-boot Environment Settings	-	-
Master Boot Record	-	-

Tabla 2: Estructura tarjeta micro SD

Por último, veamos la **secuencia de arranque** [12]**.** Consta de 4 etapas:

1. **BootROM.** Es una memoria no volátil protegida contra escritura que se encuentra dentro del chip procesador. Contiene una serie de instrucciones que sirven para inicializar los componentes *hardware* que necesita la siguiente etapa: el *Preloader*. Dichas instrucciones se ejecutan nada más arrancar la placa.
2. **Preloader.** Es un *software* basado en el *framework Second Program Loader (SPL)* que forma parte del *U-Boot*, motivo por el cual comparten entre ellos la mayor parte de su código. Entre otras funciones, el *Preloader* se encarga de inicializar la memoria SDRAM, cargar la imagen del arranque U-Boot en la SDRAM, inicializar los relojes, etc.
3. **U-Boot.** Se encarga de configurar la FPGA y cargar el *kernel* de *Linux.* Además, proporciona una consola que permite configurar los parámetros del *Device Tree Blob* y los argumentos del arranque nada más iniciar el sistema, como se aprecia en la Figura 3.

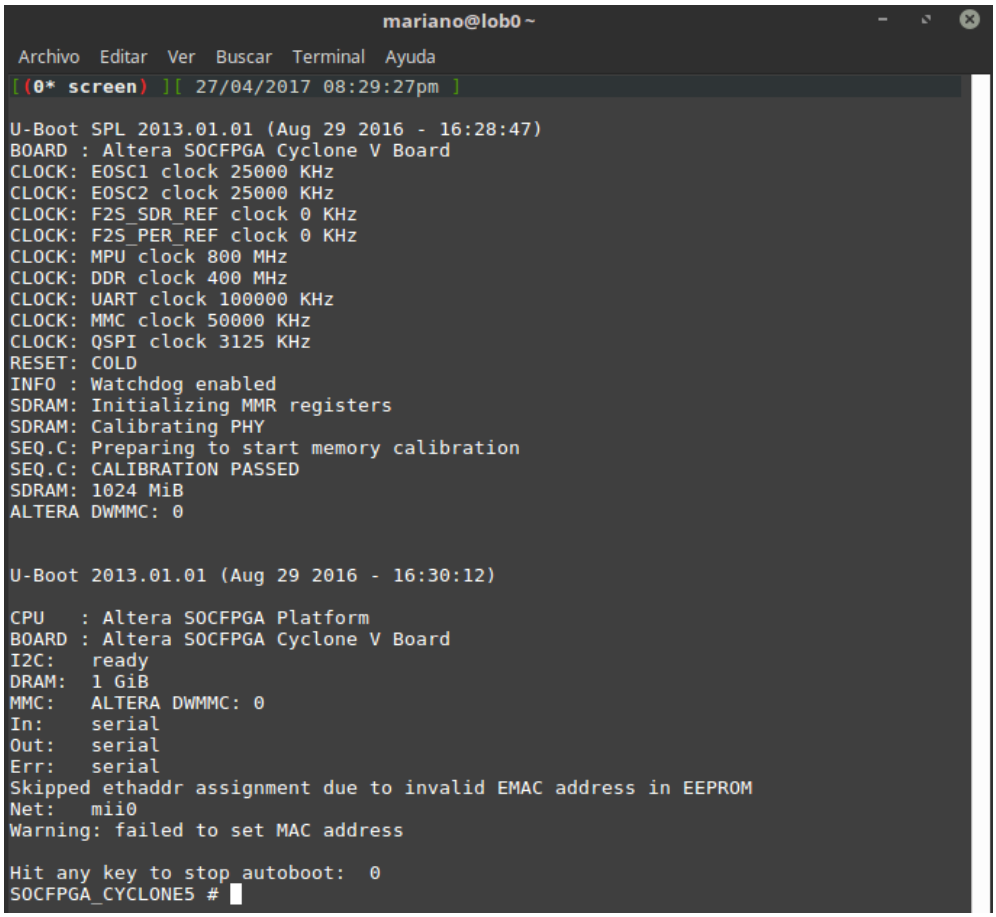


Figura 3: Captura de la consola del U-boot en el arranque de la DE1-SOC

4. **Linux**. Se encarga de ejecutar el Sistema Operativo sobre el *kernel* cargado por el U-Boot.

3.1.2 Generación de los ficheros del arranque y creación de la imagen personalizada

En este apartado vamos a ver cómo hemos generado los ficheros necesarios para crear nuestra imagen de *Debian* 8 con el *kernel* 3.13 para la DE1-SOC. Partiremos de la imagen de Linux de *Angstrom* y modificaremos los archivos necesarios para cambiar el tipo de distribución por una *Debian 8*, el *kernel* y el módulo necesario para ejecutar *OpenCL* en su versión 16.0. El resultado de este apartado será un archivo .img que podrá ser escrito en una tarjeta de memoria micro SD e insertada en la DE1-SOC preparada para funcionar.

NOTA: la generación de todos los ficheros de este apartado se ha llevado a cabo en un portátil *con Linux Mint 18 Sarah* [13]. Suponemos que el directorio de trabajo es *~/workspace*.

3.1.2.1 Preparación del equipo de compilación

Lo primero que hay que hacer es instalar las herramientas y dependencias necesarias para evitar errores en los pasos siguientes. También se necesita que nuestro Sistema Operativo esté actualizado.

La primera herramienta a instalar es el compilador de linaro [14]. Es necesario descargarla y descomprimirla:

```
$ wget https://launchpad.net/linaro-toolchain-bina-
ries/trunk/2013.10/+download/gcc-linaro-arm-linux-gnueabihf-4.8-
2013.10_linux.tar.bz2

$ tar jxfv https://launchpad.net/linaro-toolchain-bina-
ries/trunk/2013.10/+download/gcc-linaro-arm-linux-gnueabihf-4.8-
2013.10_linux.tar.bz2
```

Después hay que instalar la librería de desarrollo para *ncurses*. En *Linux Mint* (en *Ubuntu* sería igual) se hace instalando el paquete *libncurses5-dev*:

```
$ sudo apt-get install libncurses5-dev
```

Para generar el *u-boot*, se necesita la herramienta *mkimage,* que está contenida en el paquete *u-boot-tools*:

```
$ sudo apt-get install u-boot-tools
```

La herramienta *qemu-debootstrap* será útil para descargar el sistema base. Se instala mediante la instalación del paquete *qemu-user-static*, que la contiene:

```
$ sudo apt install qemu-user-static
```

Por último, se instala el device-tree-compiler y se exporta en la variable de entorno *PATH*:

```
$ sudo apt-get install device-tree-compiler

$ export PATH=`pwd`/gcc-linaro-arm-linux-gnueabihf-4.8-
2014.04_linux/bin:$PATH
```

3.1.2.2 Generación del u-boot

El U-*boot* forma parte del proyecto de *software* libre de altera que, actualmente, cuenta con una amplia lista de desarrolladores. El código puede visualizarse en el proyecto *u-boot-socfpga* del repositorio de *Github* llamado *altera-opensource* [4]. Es necesario hacer una copia de este proyecto en nuestro equipo y acceder a la rama que contiene la versión 2013.01.01:

```
$ git clone https://github.com/altera-opensource/u-boot-socfpga.git
Cloning into 'u-boot-socfpga'...
remote: Counting objects: 454675, done.
remote: Total 454675 (delta 0), reused 0 (delta 0), pack-reused 454675
```

[4] https://github.com/altera-opensource/u-boot-socfpga.git

```
Receiving objects: 100% (454675/454675), 106.36 MiB | 705.00 KiB/s,
done.
Resolving deltas: 100% (368766/368766), done.
Checking connectivity... done.

$ cd u-boot-socfpga

$ git checkout socfpga_v2013.01.01
```

Para generar el u-boot, primero hay que establecer el nombre y la ruta del compilador de linaro exportanto la variable de entorno CROSS_COMPILE:

```
$ export CROSS_COMPILE=arm-linux-gnueabihf-
```

Después y, para evitar posibles errores, hay que limpiar el directorio de trabajo:

```
$ make mrproper
```

Por último, se compila especificando el tipo de placa:

```
$ ./MAKEALL socfpga_cyclone5
Configuring for socfpga_cyclone5 board...
   text    data     bss     dec     hex filename
 227314   12568  279880  519762   7ee52 ./u-boot
  40653    3796  339200  383649   5daa1 ./spl/u-boot-spl

-------------------- SUMMARY ----------------------------
Boards compiled: 1
---------------------------------------------------------
```

En caso de no haber fallado nada se habrán generado los ficheros: *u-boot, u-boot.bin y u-boot.img*, entre otros, en la ruta: ~/workspace/u-boot-socfpga/

3.1.2.3 Compilación y configuración del kernel

El primer paso consiste en descargar las fuentes del *kernel*. No es válido un *kernel* cualquiera, sino que es necesario uno creado especialmente para ser ejecutado en SOCs como la placa utilizada en este proyecto. Al igual que antes, el repositorio *altera-opensource* contiene un proyecto de *software* libre que nos ofrece estas fuentes [15]:

```
$ git clone https://github.com/altera-opensource/linux-socfpga.git
Cloning into 'linux-socfpga'...
remote: Counting objects: 5354902, done.
remote: Total 5354902 (delta 0), reused 0 (delta 0), pack-reused
5354902
```

```
Receiving objects: 100% (5354902/5354902), 1.15 GiB | 995.00 KiB/s,
done.
Resolving deltas: 100% (4507899/4507899), done.
Checking connectivity... done.
Checking out files: 100% (57323/57323), done.
```

Al igual que antes, es necesario cambiar la rama del proyecto para descargar una versión específica. Hay que elegir la versión 3.13-rel14.0, puesto que es la última versión soportada por el A*ltera SDK for OpenCL 16.0* [16] :

```
$ cd linux-socfpga

$ git checkout socfpga-3.13-rel14.0
Checking out files: 100% (19392/19392), done.
Branch socfpga-3.13 set up to track remote branch socfpga-3.13 from
origin.
Switched to a new branch 'socfpga-3.13-rel14.0
```

A continuación, se genera el fichero de configuración ejecutando el comando:

```
$ make ARCH=arm socfpga_defconfig
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/kconfig/conf.o
  SHIPPED scripts/kconfig/zconf.tab.c
  SHIPPED scripts/kconfig/zconf.lex.c
  SHIPPED scripts/kconfig/zconf.hash.c
  HOSTCC  scripts/kconfig/zconf.tab.o
  HOSTLD  scripts/kconfig/conf
#
# configuration written to .config
#
```

Si todo ha funcionado correctamente, se habrá generado un fichero oculto llamado *.config*. La herramienta *menuconfig* nos permite modificarlo de una forma segura. Se tiene que habilitar la opción CONFIG_FHANDLE. Para ello, se ejecuta la herramienta con el comando:

```
$ make ARCH=arm menuconfig
```

Se selecciona la opción *General Setup* y se pulsa *enter*, como indica la Figura 4.

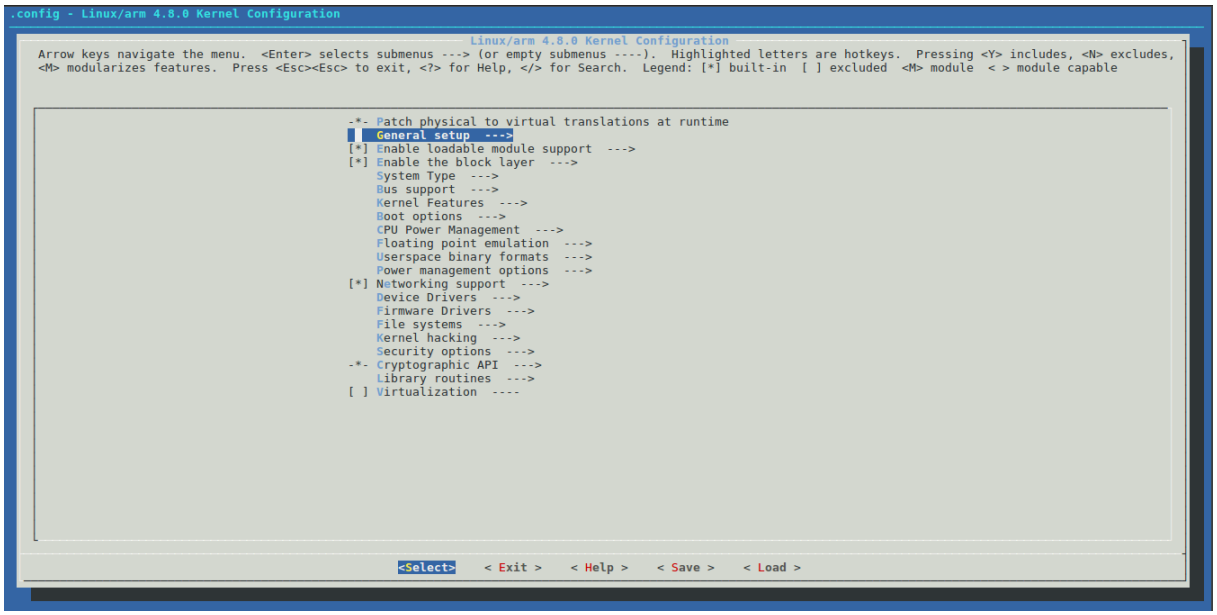


Figura 4: menuconfig, paso 1

Se marca con la barra espaciadora la opción *Open by fhandle syscalls,* como muestra la Figura 5.

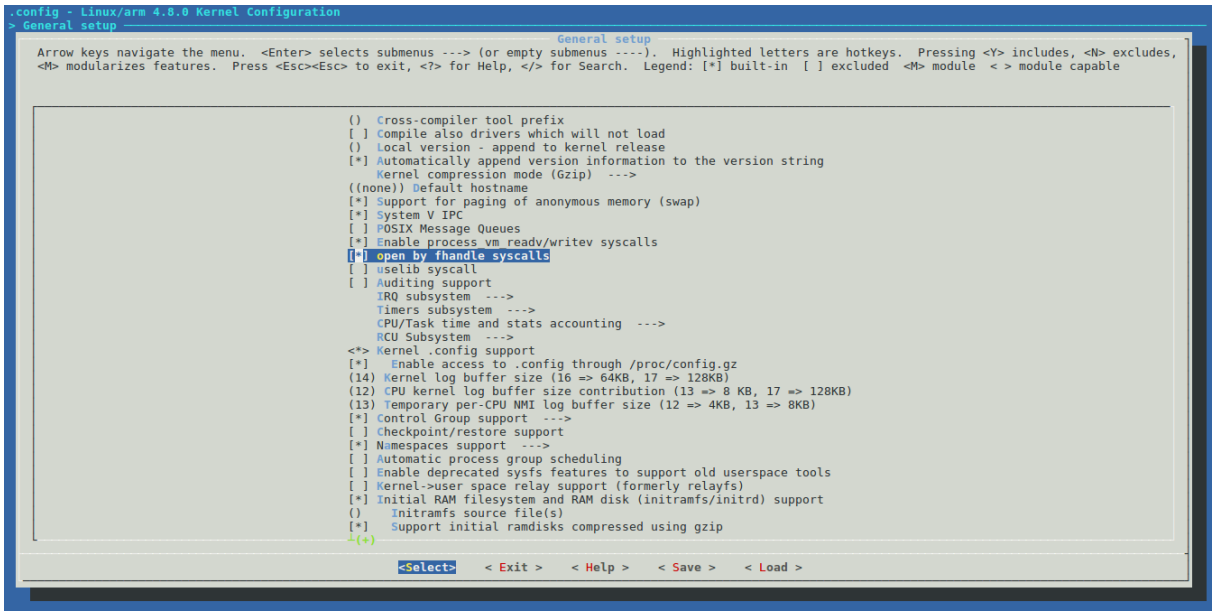


Figura 5: menuconfig, paso 2

Después se guardan los cambios seleccionando la opción “Save”, con el nombre .config, como se observa en la Figura 6.

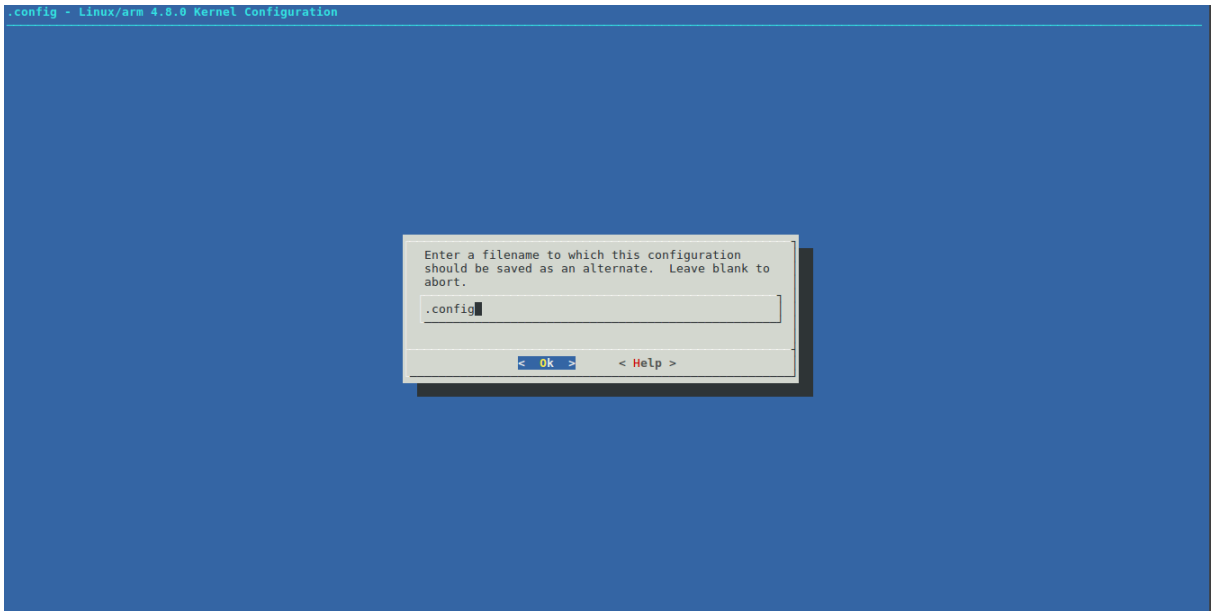


Figura 6: menuconfig, paso 3

Como veremos en el apartado relacionado con la compilación e instalación de *OpenCL,* es necesario activar el CMA (*Contiguous Memory Allocator*) para poder cargar el *driver* de *OpenCL* y poder así ejecutar nuestros programas. Para ello, se modifica el fichero *~/workspace/Linux-socfpga/arch/arm/configs/socfpga_defconfig*, añadiendo al final:

```
CONFIG_MEMORY_ISOLATION=y
CONFIG_CMA=y
CONFIG_DMA_CMA=y
CONFIG_CMA_DEBUG=y
CONFIG_CMA_SIZE_MBYTES=512
CONFIG_CMA_SIZE_SEL_MBYTES=y
CONFIG_CMA_ALIGNMENT=8
CONFIG_CMA_AREAS=7
```

En este punto ya se puede compilar el *kernel* 3.13. Para ello, se ejecuta el siguiente comando y se espera a que termine:

```
$ make -j8 ARCH=arm uImage LOADADDR=0x8000
...
Image Name:   Linux-3.13
Created:      Wed Mar 22 17:37:42 2017
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    4157600 Bytes = 4060.16 kB = 3.96 MB
Load Address: 00008000
Entry Point:  00008000
Kernel: arch/arm/boot/uImage is ready
```

```
$ make ARCH=arm dtbs
  CHK     include/config/kernel.release
  CHK     include/generated/uapi/linux/version.h
  CHK     include/generated/utsrelease.h
  CHK     include/generated/bounds.h
  CHK     include/generated/timeconst.h
  CHK     include/generated/asm-offsets.h
  CALL    scripts/checksyscalls.sh
  DTC     arch/arm/boot/dts/socfpga_arria5_socdk.dtb
  DTC     arch/arm/boot/dts/socfpga_arria10_socdk_nand.dtb
  DTC     arch/arm/boot/dts/socfpga_arria10_socdk_qspi.dtb
  DTC     arch/arm/boot/dts/socfpga_arria10_socdk_sdmmc.dtb
  DTC     arch/arm/boot/dts/socfpga_arria10_swvp.dtb
  DTC     arch/arm/boot/dts/socfpga_cyclone5_mcvevk.dtb
  DTC     arch/arm/boot/dts/socfpga_cyclone5_socdk.dtb
  DTC     arch/arm/boot/dts/socfpga_cyclone5_de0_sockit.dtb
  DTC     arch/arm/boot/dts/socfpga_cyclone5_sockit.dtb
  DTC     arch/arm/boot/dts/socfpga_cyclone5_socrates.dtb
  DTC     arch/arm/boot/dts/socfpga_cyclone5_trcom.dtb
  DTC     arch/arm/boot/dts/socfpga_cyclone5_vining_fpga.dtb
  DTC     arch/arm/boot/dts/socfpga_vt.dtb

$ make -j8 ARCH=arm modules
...
```

Si todo ha ido bien, se habrán generado los siguientes dos ficheros:

- **zImage** en la ruta: ~/workspace/linux-socfpga/arch/arm/boot
- **socfpga_cyclone5_socdk.dtb** en la ruta:

  ~/workspace/linux.socfpga/arch/arm/boot/dts/

3.1.2.4 Creación del sistema raíz

Lo último paso consiste en generar el sistema raíz. En este proyecto se ha elegido utilizar *Debian 8* por ser un Sistema Operativo muy estable, de buen rendimiento, actualizado y con una gran comunidad de usuarios detrás de él. Otras alternativas igualmente válidas habrían sido *Ubuntu* [17] o *CentOS* [18].

En este punto del proyecto se va a utilizar la herramienta *qemu-debootstrap* que se ha instalado al principio. Se crea un directorio llamado *rootfs* y se ejecuta el siguiente comando. La opción
--*arch* sirve para especificar la arquitectura de la máquina donde se ejecutará el sistema:

```
$ sudo qemu-debootstrap --no-check-gpg --arch=armhf jessie rootfs
ftp://ftp.debian.org/debian/
```

Por último, se instalan las herramientas y se editan los ficheros que se crea conveniente mediante el comando *chroot*. En nuestro caso, se han modificado los siguientes ficheros:

```
$ chroot rootfs
```

- Se instala un servidor de **SSH**: paquete ssh-server. Después, para aumentar la velocidad de la conexión, se edita el fichero */etc/ssh/sshd_config* añadiendo:

  ```
  UsePAM no
  UseDNS no
  ```

- Se configura la **contraseña** del usuario *root* con el comando *passwd* y se establece al valor **debiandebian**.

- Se modifica el fichero */etc/interfaces* para que la IP se asigne automáticamente por DHCP y que lo haga **una vez se hayan arrancado todos los servicios**. Esto es importante porque parece ser que, si la red se intenta auto configurar en medio del arranque, como suele ser habitual, cuando el arranque finaliza no se ha asignado una IP. El contenido de dicho fichero es el siguiente:

  ```
  # interfaces(5) file used by ifup(8) and ifdown(8)
  # Include files from /etc/network/interfaces.d:
  source-directory /etc/network/interfaces.d

  # DCHP dado por el router
  allow-hotplug eth0
  iface eth0 inet dhcp
  ```

3.1.2.5 Creación del fichero imagen con la Debian 8 y kernel 3.13

El resultado de este apartado es un fichero de extensión .img, que permitirá de una forma muy sencilla, instalar una *Debian 8* con el *kernel* 3.13 en la DE1-SOC. Es un fichero similar a los que se pueden descargar desde Internet, por ejemplo, los que vimos en el Estado del Arte.

Como hemos comentado con anterioridad, partiremos de la imagen de Linux de *Angstrom,* que estudiamos en el Estado del Arte, y solamente se modificarán aquellos ficheros que sean necesarios para cambiar el *kernel* y el Sistema Operativo. Para ello, se copia la imagen de *Angstrom* en una tarjeta micro SD con el comando *dd*, se conecta a un PC y se montan las particiones 1 y 2.

En la **partición 1** se copian los siguientes ficheros para cambiar el *kernel* de la distribución de *Angstrom* por el *kernel* del proyecto libre que se ha compilado compilado:

- ~/workspace/linux-socfpga/arch/arm/boot/**zImage**

- ~/workspace/linux-socfpga/arch/arm/boot/dts/**socfpga_cyclone5_socdk.dtb** con el nombre **socfpga.dtb**

Por otro lado, se copia el contenido del directorio *~/workspace/rootfs* a la **partición 2**, eliminado previamente todo su contenido. Esto cambiará el sistema base de *Angstrom* por La *Debian 8*.

Para evitar repetir estos pasos es conveniente crear un fichero de extensión .img. De esta manera, nuestra imagen podrá ser compartida y cualquier persona podrá instalarla en una placa DE1-SOC. Además, en caso de error, podremos volver a dejar la placa con una instalación limpia de una manera rápida y sencilla. Nosotros utilizamos el comando *dd:*

```
$ sudo dd if=/dev/mmcblk0 of=~/workspace/Linux_debian8_3.13_maria-
noh.img bs=1M status=progress
```

3.1.3 Compilación e instalación del OpenCL Linux Kernel Driver

Para poder ejecutar *OpenCL* en la FPGA de la DE1-SOC, la imagen que se acabada de crear en el apartado anterior debe contener las fuentes de *OpenCL* y un *driver* de *OpenCL* específico de la versión del *kernel* que se ha instalado.

Las fuentes están incluidas en el fichero aocl-rte-.arm32.tgz, que se puede obtener de la web oficial de altera. Para ello hay que descargar el fichero *Altera Runtime Environment for OpenCL Linux Cyclone V SoC TGZ* [5]*,* versión 16.0. Es necesario establecer en la variable del entorno del *PATH* la ruta del compilador de linaro y en la variable *CROSS_COMPILE* el nombre del ejecutable del compilador:

```
$ export PATH=`pwd`/gcc-linaro-arm-linux-gnueabihf-4.8-
2014.04_linux/bin:$PATH

$ export CROSS_COMPILE=arm-linux-gnueabihf-
```

Una vez descargado y descomprimido el fichero *aocl-rte-.arm32.tgz,* en la ruta *aocl-rte-.arm32/board/c5soc/driver,* hay que limpiar el directorio de trabajo y compilar:

```
$ make clean
$ make
```

Si todo ha ido bien, se habrá creado el *driver* de *OpenCL:* **aclsoc_drv.ko**.

Es necesario crear un *script* que establezca en ciertas variables de entorno la ruta de las fuentes y del driver de *OpenCL*. Este *script* tendrá que ser lanzado cada vez que se encienda la placa y se quieran ejecutar programas de *OpenCL*. Su contenido es el siguiente:

```
export ALTERAOCLSDKROOT=/root/aocl-rte-.arm32
export AOCL_BOARD_PACKAGE_ROOT=$ALTERAOCLSDKROOT/board/c5soc
export PATH=$ALTERAOCLSDKROOT/bin:$PATH
export LD_LIBRARY_PATH=$ALTERAOCLSDKROOT/host/arm32/lib:$LD_LI-
BRARY_PATH
insmod $AOCL_BOARD_PACKAGE_ROOT/driver/aclsoc_drv.ko
```

[5] http://dl.altera.com/opencl/16.0/?edition=standard

El último paso consiste en actualizar estos ficheros en la imagen (extensión .img) que se ha preparado en el apartado anterior. Hay que copiar las fuentes y el *driver* en el directorio del sistema base, *rootfs:*

- El directorio *aocl-rte-.arm32/* en la ruta *rootfs/root.*
- El *driver aclsoc_drv.ko* en la ruta *rootfs/root/aocl-rte-.arm32/c5soc/driver*.
- El *script init_opencl.sh* en la ruta *rootfs/root*.

Se genera la imagen en formato .img, para facilitar su instalación:

```
$ sudo dd if=/dev/mmcblk0 of=~/work-
space/Linux_debian8_3.13_OpenCL_RTE_16_marianoh.img bs=1M status=pro-
gress
```

La Figura 7 es una captura de pantalla de la imagen que se acaba de generar. En ella se muestra el nombre de la distribución y el *kernel* que se ha compilado:

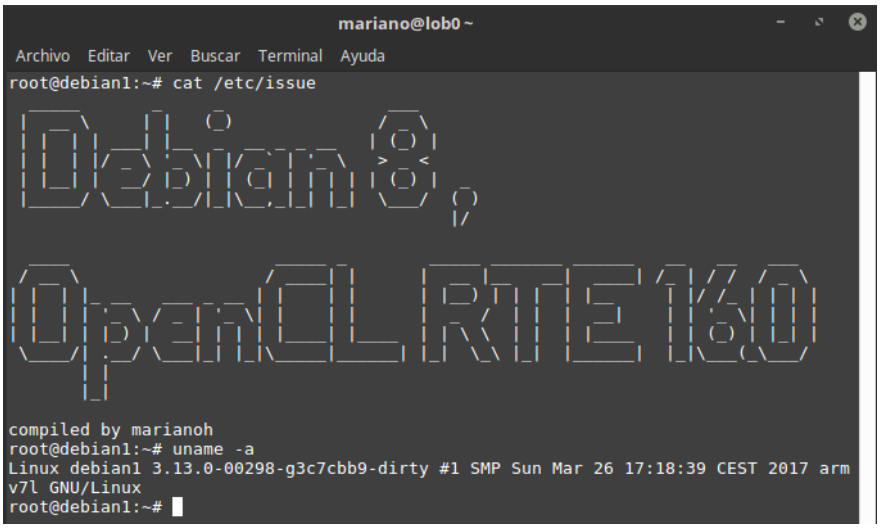


Figura 7: Captura de pantalla de la imagen de Linux generada

En las siguientes capturas de pantalla se muestra cómo se han ejecutado 2 códigos de ejemplo de *OpenCL* satisfactoriamente. La Figura 8 es un ejemplo que calcula la suma de 2 vectores generados de forma aleatoria, y la Figura 9 un hola mundo que emplea dos hilos.

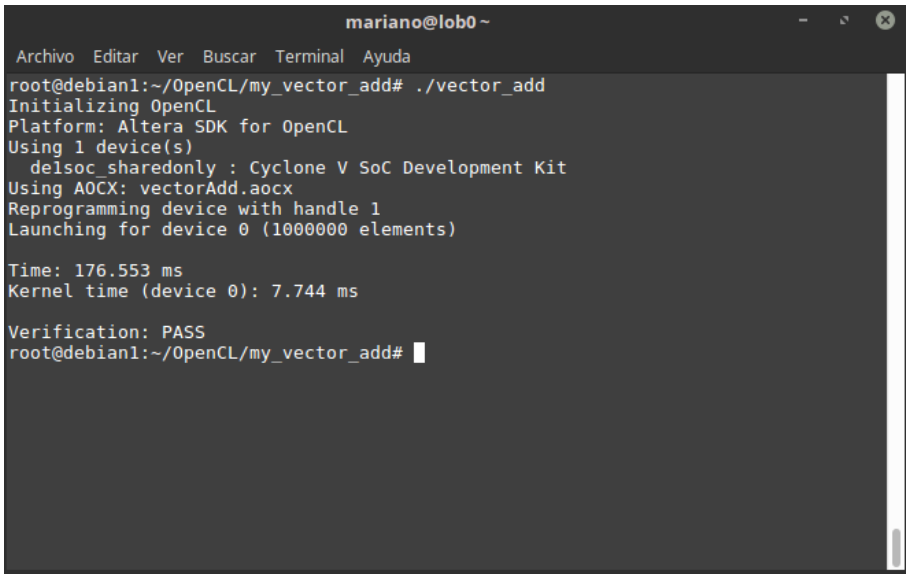


Figura 8: Ejemplo vector_add de OpenCl ejecutado sobre una DE1-SOC

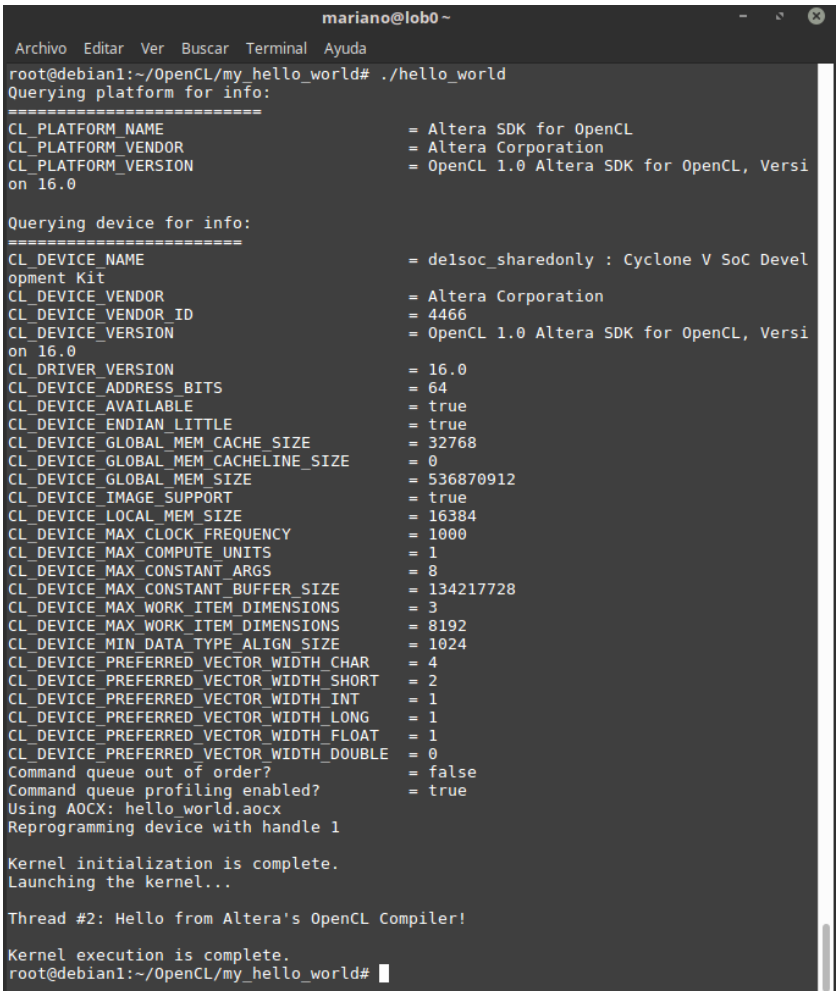


Figura 9: Ejemplo hello_world de OpenCL ejecutado sobre una DE1-SOC

3.2 Creación del *clúster*

En este apartado vamos a ver la estructura del *clúster* que se ha construido en función del hardware disponible y cómo se ha configurado. La Figura 10 representa el diagrama de las conexiones entre los elementos del clúster, que son los siguientes:

- **4 placas *Altera Cyclone V DE1-SOC***. Este tipo de placas, consideradas de bajo consumo y de bajo coste, han sido donadas dentro del *University Program* de Altera[6].
- **2 enrutadores de 4 puertos *Fast Ethernet***. No se tratan de *routers* de alto rendimiento, sino que son los típicos que proporcionan los Proveedores de servicios de Internet (ISP en su sigla en inglés). En nuestro caso son un *Amper Xavi 7868r* de *Movistar* y un *TD5130* de *ONO.* Necesitamos conectarlos entre sí en la misma red local puesto que necesitamos que los nodos esclavos se puedan comunicar con el maestro. Esto hace un total de 5 conexiones (el nodo maestro más 4 nodos esclavos). Por ser más moderno, el *TD5130* será el *router* maestro y, por lo tanto, será el servidor *DHCP*. Por otro lado, el *Amper Xavi 7868r* será el *router* secundario y será cliente de dicho servidor *DHCP.*
- **6 cables Fast Ethernet**, cuyo ancho de banda es de 100Mbps (unos 12,5 MBps).
- **Ordenador portátil**. Su finalidad es ser el nodo maestro, es decir, el nodo encargado de lanzar las ejecuciones del código a los nodos y juntar los resultados de cada uno de ellos.

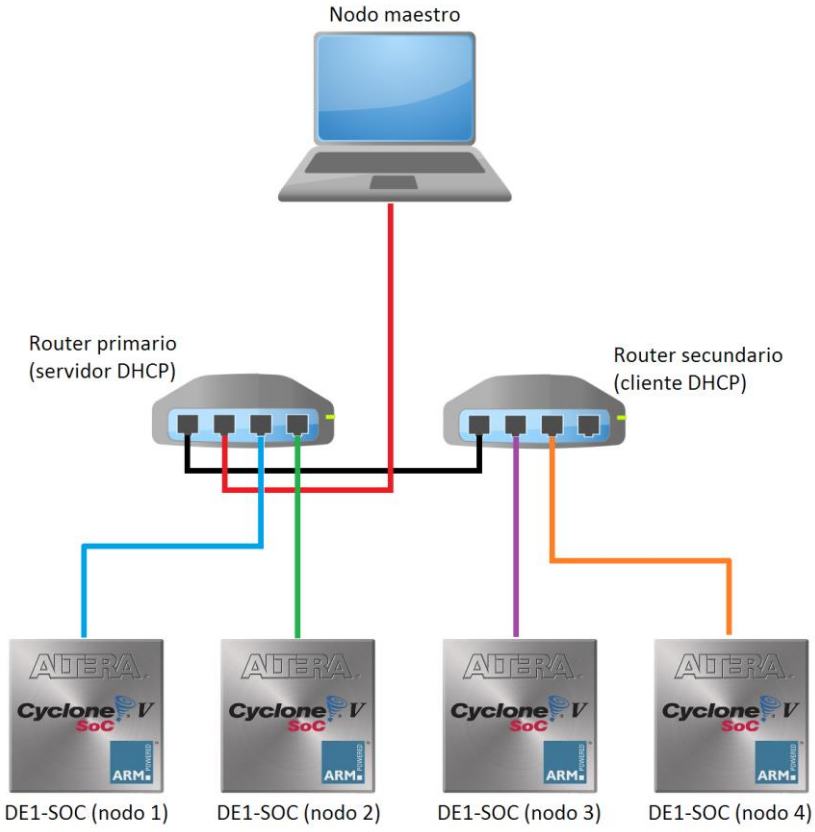


Figura 10: Estructura del clúster con 4 DE1-SOCs

[6] https://www.altera.com/support/training/university/overview.html

Una vez conectadas las placas, hay que asegurarse de que el fichero */etc/network/interfaces* configure la red por *DHCP* y que la configuración de la red se haga en el último paso del arranque de la placa, como vimos en la sección 3.1.2.4 Creación del sistema raíz. El contenido del fichero anterior debería ser el siguiente:

```
# interfaces(5) file used by ifup(8) and ifdown(8)
# Include files from /etc/network/interfaces.d:
source-directory /etc/network/interfaces.d

# DCHP portatil o router
allow-hotplug eth0
iface eth0 inet dhcp
```

El último paso es hacer que la conexión entre el nodo maestro y cualquiera de los nodos esclavos, y viceversa, se haga **sin requerir contraseña**. Esto es necesario para poder enviar los datos y resultados, así como ejecutar comandos de manera remota sin necesidad de que el usuario ingrese la contraseña por cada petición. Para ello, hay que:

- Generar una clave pública sin introducir *passphrase* en cada uno de los nodos (maestro y esclavos).
- En el nodo maestro: añadir la clave pública de cada una de las placas.
- En cada una de las placas: añadir la clave pública del nodo maestro.

Como ejemplo, vamos a ver los comandos que habría que ejecutar para configurar el nodo maestro y uno de los nodos esclavos:

```
marian0@master$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/marian0/.ssh/id_rsa):
[PULSAMOS ENTER]
Enter passphrase (empty for no passphrase): [PULSAMOS ENTER]
Enter same passphrase again: [PULSAMOS ENTER]
Your identification has been saved in /home/marian0/.ssh/id_rsa.
Your public key has been saved in /home/marian0/.ssh/id_rsa.pub.
The key fingerprint is:
33:c3:ee:af:f5:e5:19:31:11:c5:de:96:2f:f2:35:f9 marian0@master

marian0@master$ ssh-copy-id -i ~/.ssh/id_rsa.pub root@[IP_nodo1]
[INTRODUCIMOS LA PASSWORD DEL USUARIO ROOT DE LA PLACA 1 (debiande-
bian)]
Now try logging into the machine, with "ssh 'remote-host'", and check
in:
.ssh/authorized_keys
to make sure we haven't added extra keys that you weren't expecting.

root@nodo1$ ssk-keygen
```

```
Generating public/private rsa key pair.
Enter file in which to save the key (/home/root/.ssh/id_rsa): [PULSA-
MOS ENTER]
Enter passphrase (empty for no passphrase): [PULSAMOS ENTER]
Enter same passphrase again: [PULSAMOS ENTER]
Your identification has been saved in /home/root/.ssh/id_rsa.
Your public key has been saved in /home/root/.ssh/id_rsa.pub.
The key fingerprint is:
33:b7:aa:ff:90:90:22:21:30:dd:ed:07:5f:ff:02:19 root@nodo1

root@nodo1$ ssh-copy-id -i ~/.ssh/id_rsa.pub mariano@[IP_master]
[INTRODUCIMOS LA PASSWORD DEL USUARIO MARIANO DEL NODO MASTER]
Now try logging into the machine, with "ssh 'remote-host'", and check
in:
.ssh/authorized_keys
to make sure we haven't added extra keys that you weren't expecting.
```

4. Capítulo IV: Experimentos con *OpenCL*

El objetivo de este apartado es comparar el rendimiento del *clúster* que acabamos de construir con el de una *Workstation*. Para ello, se han elegido un conjunto de 5 algoritmos que procesan imágenes, al que denominaremos *suite*, que ha sido considerado de suficiente complejidad. Dichos algoritmos serán lanzados en el *clúster* con 1, 2, 3 y 4 nodos y serán programados en *OpenCL* [19] utilizando el *Intel FPGA SDK for OpenCL* versión 16.0 [20]. Después, se implementarán en el lenguaje C y serán ejecutados tanto en una DE1-SOC de manera local, como en una *Workstation.* Por último, se compararán los resultados entre sí.

4.1 Arquitectura del Altera SoC *OpenCL*

Un proyecto de *Altera OpenCL* para SoCs consta de dos programas: el ***kernel*** y el ***host***. El *kernel* es el código que es ejecutado en la FPGA del SoC. Se trata de un fichero de extensión **AOCX** generado con los programas *Quartus* e *Intel FPGA SDK for OpenCL.* El *host* es el programa que es ejecutado en el procesador *ARM Cortex A9 Dual-Cor*e de la placa. Es de extensión **ELF**, como cualquier ejecutable de UNIX/LINUX, y es generado con el compilador estándar de C (gcc). La Figura 11, obtenida del manual "*DE1-SOC OpenCL v2.0*" y modificada por nosotros, ejemplifica esta idea.

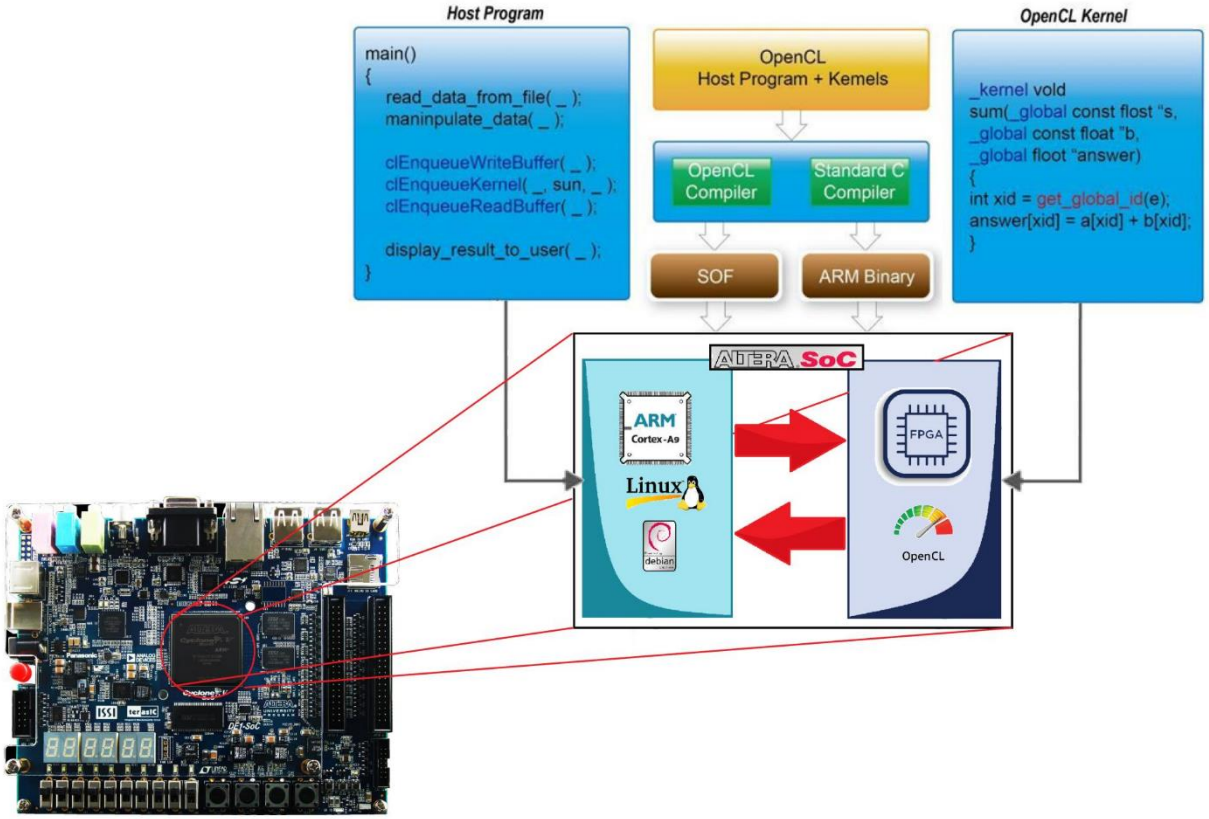


Figura 11: Arquitectura Altera SoC OpenCL

Estos ficheros se generan en una máquina distinta a las placas. Esto se conoce como **compilación cruzada**. Para nuestro proyecto y dado que la compilación de *OpenCL* requiere de un *hardware* con mucha memoria RAM y muchos accesos a disco, se ha decidido utilizar una *Workstation* cuyo acceso ha sido facilitado por el Departamento de Arquitectura de Computadores y Automática (DACYA) de nuestra facultad. Esta *Workstation* cuenta con 32 *cores*, 32GB de RAM y un SSD de alto rendimiento de *Intel*. El *software* que ha sido necesario instalar en ella ha sido el siguiente:

- *Quartus Prime Design Suite Release 16.0.*
- *Intel FPGA SDK for OpenCL* versión 16.0.
- *Altera SoC EDS.*
- *DE1-SOC OpenCL Board Support Package (BSP)*

Los comandos necesarios para realizar una compilación del programa *kernel* y del programa host en dicha *Workstation* son, respectivamente, los siguientes:

```
$ aoc device/programa_kernel.cl –sw-dimm-partition -o bin/kernel.aocx
$ make
```

4.2 Script de lanzamiento de los nodos esclavos

Como veremos más adelante, el nodo maestro mandará ejecutar estos algoritmos al *clúster* a través de un lanzador escrito en *bash.* Este lanzador invocará de forma remota el programa y los argumentos a ejecutar en cada placa a través de un *script* que tendrá cada nodo. El contenido de ese *script* es el siguiente:

```
#!/bin/bash

function get_time(){
  echo `date +%s%N`
}

export ALTERAOCLSDKROOT=/root/aocl-rte-16.0.0-1.arm32
export AOCL_BOARD_PACKAGE_ROOT=$ALTERAOCLSDKROOT/board/c5soc
export PATH=$ALTERAOCLSDKROOT/bin:$PATH
export LD_LIBRARY_PATH=$AL-
TERAOCLSDKROOT/host/arm32/lib:/root/extlibs/lib:$LD_LIBRARY_PATH

EXECUTABLEPATH="/tmp"
MASTER="192.168.1.33"

mkdir -p $EXECUTABLEPATH/out

echo "running" > state
time0=$(get_time)
$EXECUTABLEPATH/$2 $3 $4 $5 $6 $7 $8> DEBUG_$2.log
```

```
time1=$(get_time)
scp $EXECUTABLEPATH/out/* mariano@$MASTER:$1
time2=$(get_time)
rm -rf /tmp/*

let execTime=$time1-$time0
let scpTime=$time2-$time1

echo "$execTime" > $2.log
echo "$scpTime" >> $2.log

echo "finish" > state
```

Como se puede observar, lo primero que hace es exportar las variables del entorno que especifican la ruta del *driver* de *OpenCL* que instalamos en la sección 3.1.3 Compilación e instalación del OpenCL Linux Kernel Driver y las librerías externas que se van a utilizar. Después se configuran la ruta de trabajo y la dirección IP del nodo maestro. Por último, se ejecuta el programa con sus argumentos redirigiendo la salida a un fichero de *log*, se actualiza el estado del nodo y se escriben los tiempos de ejecución y copia de resultados en otro dichero de *log.* El nodo maestro será el encargado de leer el estado de cada nodo, contenido en el fichero *state* (explicado más abajo), para llevar a cabo la sincronización del *clúster* y de leer el fichero de *log* que contiene los tiempos del nodo para mostrarlos junto con el resto de tiempos.

El fichero de texto *state* contiene el estado actual del nodo: *idle* (está listo para ejecutar un programa), *running* (está ejecutando un programa) o *finished* (ha terminado de ejecutar un programa y está a la espera de que el nodo maestro sea consciente de esto y cambie el estado a *idle*).

4.3 Experimento I: Detección de bordes con el operador de Laplace

El primero de los algoritmos que forman parte de esta *suite* se encarga de aplicar un filtro sobre una imagen para detectar los bordes que aparecen en ella. Más concretamente aplica el operador de *Laplace* [21]. Sin entrar mucho en detalle, este operador aplica una matriz 3x3 de unos coeficientes obtenidos mediante la diferencia central, que dice lo siguiente:

$$\frac{df}{dx} \approx \frac{f(x+1) - f(x-1)}{2}$$

Extendiendo este resultado a matrices de 2 dimensiones se obtiene:

$$\frac{d^2 f}{dx^2} \approx [f(x+1) - f(x)] - [f(x) - f(x-1)] = f(x+1) - 2f(x) + f(x-1)$$

Que puede ser aproximada con la matriz de coeficientes [1, -2, 1]. Creando a partir de ella las matrices de las direcciones X e Y, y sumándolas entre sí, se obtiene la matriz que representa los coeficientes de Laplace:

$$\begin{pmatrix} 0 & 0 & 0 \\ 1 & -2 & 1 \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 1 & 0 \\ 0 & -2 & 0 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Después de varios análisis previos, se ha decido no utilizar estos coeficientes porque generan demasiado ruido en el resultado. Para solucionar este problema, se ha optado por utilizar la siguiente variación:

$$\begin{pmatrix} -0{,}5 & 0 & -0{,}5 \\ -1 & 4 & -1 \\ -0{,}5 & 0 & -0{,}5 \end{pmatrix} + \begin{pmatrix} -0{,}5 & -1 & -0{,}5 \\ 0 & 4 & 0 \\ -0{,}5 & -1 & -0{,}5 \end{pmatrix} = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

Para implementar este filtro en *OpenCL*, partiremos del código de ejemplo "*Sobel Filter Design Example*" que se puede descargar desde la web de *Altera* [22] y lo modificaremos para evitar que utilice *OpenGL* y la librería *SDL* para mostrar el resultado del filtro en tiempo real. Esta modificación se justifica en el hecho de que las placas DE1-SOC no cuentan con tarjeta gráfica y, por lo tanto, son incapaces de ejecutar código escrito en *OpenGL* o librerías gráficas como la librería *SDL* utilizada por éste ejemplo. En nuestro caso, el algoritmo aplicará el filtro de Laplace sobre una imagen pasada como parámetro y finalizará.

La entrada es una imagen en color en RGB, con una profundidad de 24 bits (255 colores) y de formato PPM. La salida será también una imagen en formato PPM, pero será bicolor: blanco para los bordes y negro para el resto. La Figura 12 ilustra el resultado de aplicar el filtro con el operador de Laplace (imagen inferior) sobre una imagen de 1920 x 1080 píxeles (arriba).

Figura 12: Imagen de resultado detección de bordes de Laplace

4.3.1 Código del kernel de *OpenCL*

A continuación, vamos a ver el código del *kernel,* que es el código de *OpenCL* que se ejecutará en la FPGA de cada placa. Tanto para este, como para el resto de programas de la *suite*, solo se mostrará el código del *kernel*. El resto del código podrá encontrarse en mi *Github* [23].

Recibe 3 parámetros de entrada y uno de salida:

- ***frame_in (entrada):*** es un puntero al *array* que contiene la imagen en formato PPM que se ha leído.
- ***iterations (entrada):*** número de vueltas que tiene que ejecutarse el bucle principal. Coincide con el alto de la imagen multiplicado con el ancho.
- ***thresold (entrada):*** umbral a partir del cual un pixel se considerará borde o no. Sirve para generar una imagen con menos ruido. Por norma general, cuanto más bajo sea este valor, más ruido tendrá la imagen resultado. Sin embargo, si se toma un valor demasiado alto, muchos bordes pueden ser considerados como ruido y, por lo tanto, no salir en la

imagen resultado. En nuestro caso utilizamos un valor de 32, que a tenor de las diferentes experimentaciones realizadas guarda un buen compromiso entre estos dos efectos mencionados.

- ***frame_out (salida):*** es un puntero al array que contiene la imagen resultante.

```
__kernel void laplacian(
     global unsigned int * restrict frame_in,
     global unsigned int * restrict frame_out,
     const int iterations,
     const unsigned int threshold) {
```

Los coeficientes no se toman como parámetro de entrada, sino que se incluyen en el propio código del *kernel*. Se podría haber pasado como parámetro, pero puesto que hemos decidido fijarlo según la aproximación que hemos visto antes, no es necesario:

```
    // Filter coefficients
    int Gx[3][3] = {{-1,-1,-1},{-1,8,-1},{-1,-1,-1}};
```

El bucle principal aplica la matriz de coeficientes Gx a cada píxel. Con la directiva *#pragma unroll* se consigue computar una aplicación de dicha matriz por cada ciclo, aumentando considerablemente el rendimiento frente a la versión iterativa en C. Para poder utilizar esta directiva es necesario que se conozcan los límites de los bucles en **tiempo de compilación.** Esta es la razón por la que el alto y ancho de la imagen son constantes globales (*ROWS* y *COLS*) y, por lo tanto, es necesario compilar varios *kernel*s con tamaños de imagen distintos:

```
    // Pixel buffer of 2 rows and 3 extra pixels
    int rows[2 * COLS + 3];
    // The initial iterations are used to initialize the pixel buffer.
    int count = -(2 * COLS + 3);

    while (count != iterations) {

        #pragma unroll
        for (int i = COLS * 2 + 2; i > 0; --i) {
            rows[i] = rows[i - 1];
        }
        rows[0] = count >= 0 ? frame_in[count] : 0;

        int x_dir = 0;

        #pragma unroll
        for (int i = 0; i < 3; ++i) {
            #pragma unroll
            for (int j = 0; j < 3; ++j) {
```

La imagen de entrada se pasa como un puntero y, al ser leída en el *Main*, se construye poniendo en orden cada componente de color (roja, verde y azul) de cada píxel. Antes de aplicar la matriz de los coeficientes hay que obtener la componente de color de cada píxel. Para mayor eficiencia, lo hacemos con un simple desplazamiento y con una *AND* lógica:

```
unsigned int pixel = rows[i * COLS + j];
unsigned int b = pixel & 0xff;
unsigned int g = (pixel >> 8) & 0xff;
unsigned int r = (pixel >> 16) & 0xff;
```

Otra consideración importante es que se realiza una aproximación de los colores de RGB (rojo, verde, azul) a LUMA para aumentar el rendimiento. El formato LUMA representa los colores de una imagen con una escala de grises en función de la luminosidad. En este caso, se ha optado por una aproximación basada en enteros. Una vez obtenida esta aproximación, se multiplica el píxel por la matriz:

```
            // RGB -> Luma conversion approximation
            unsigned int luma = r * 66 + g * 129 + b * 25;
            luma = (luma + 128) >> 8;
            luma += 16;

            x_dir += luma * Gx[i][j];
        }
    }
```

Por último, se comprueba si el píxel está por debajo o por encima del umbral definido por la variable de entrada *threshold*. Si un píxel está por encima de este valor será un borde y, por lo tanto, se pintará de blanco en la imagen resultado. En caso contrario se pintará de negro:

```
        int temp = abs(x_dir);
        unsigned int clamped;

        if (temp > threshold) {
            clamped = 0xffffff;
        }
        else {
            clamped = 0;
        }

        if (count >= 0) {
            frame_out[count] = clamped;
        }
        count++;
    }
}
```

4.3.2 Lanzador del algoritmo en el *clúster*

En este apartado vamos a ver cómo hemos ejecutado este algoritmo en el *clúster* variando el número de placas. En este caso, la forma de paralelizar el algoritmo es bastante sencilla: dividimos la imagen a procesar en N fragmentos, siendo N el número de placas; mandamos ejecutar en cada placa una porción de la imagen y, al final, juntamos los resultados. La configuración de las placas se almacena en un fichero que contiene la lista de las direcciones IP de los nodos del clúster.

Para analizar los resultados de ejecutar los algoritmos en el *clúster* con distinto número de nodos, hemos decidido emplear una función que permite obtener el tiempo en nanosegundos y utilizarla para calcular el tiempo de ejecución de distintos fragmentos de código, simplemente realizando una diferencia entre 2 valores almacenados. En este caso, hemos decidido medir los tiempos de:

- Dividir la imagen de entrada en fragmentos.
- Copiar los datos entre el nodo maestro y el nodo esclavo i-ésimo correspondiente.
- Ejecutar el algoritmo en el nodo i-ésimo.
- Copiar los resultados entre el nodo i-ésimo y el nodo maestro.
- Juntar los resultados de cada nodo en una única imagen.

El pseudocódigo sería el siguiente:

```
Parámetros de entrada:
   1. Ejecutable del programa. Extensión ELF.
   2. Kernel de OpenCL compilado. Extension AOCX.
   3. Imagen de entrada. Extensión PPM.
   4. Ancho de la imagen (en píxeles).
   5. Alto de la imagen (en píxeles).

Inicio
      CONFIG=”de1soc-cluster.conf”
      PATH=”/tmp”

      Leer las direcciones IP de las placas del fichero CONFIG
      NUM_NODOS=número de líneas leídas de CONFIG

      Tiempo0=tomar_tiempo()
      Dividir la imagen de entrada verticalmente en NUM_NODOS trozos
      Tiempo1=tomar_tiempo()

      Para i=0 hasta NUM_NODOS hacer (en paralelo):
            Copiar el ejecutable del programa, el fichero del kernel AOCX
            y el trozo de imagen i-ésimo al nodo (i).

            Mandar ejecutar en remoto al nodo (i).
      Fin
```

```
        Esperar a que todos los nodos finalicen
        Tiempo2=tomar_tiempo()
        Juntar los NUM_NODOS resultados en una única imagen PPM
        Tiempo3=tomar_tiempo()

        Tiempo_dividir=Tiempo1-Tiempo0
        Tiempo_ejecutar=Tiempo2-Tiempo1
        Tiempo_juntar=Tiempo3-Tiempo2
        Tiempo_total=Tiempo3-Tiempo0
        Mostrar tiempos
Fin
```

Hemos decidido utilizar el lenguaje de *scripting bash* puesto que nos permite ejecutar comandos de forma remota, así como copiar ficheros entre los nodos de forma sencilla y sin solicitar la contraseña. El código completo puede verse en este mismo documento en el Anexo I.

4.3.3 Resultados

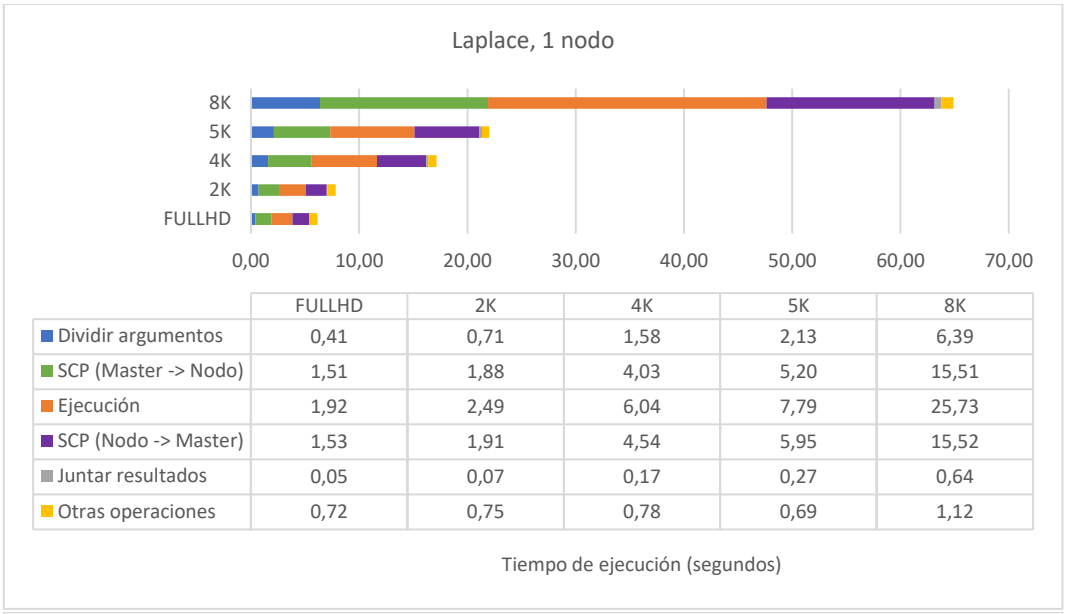
En este apartado vamos a comparar los resultados del tiempo de ejecución del algoritmo de detección de bordes con el operador de Laplace sobre el *clúster* con uno, dos, tres y cuatro nodos. Hemos realizado las pruebas con 5 imágenes de entrada de distintas resoluciones:

- **FULLHD**: 1920 x 1080 píxeles de alto y ancho, respectivamente.
- **2K**: 2560 x 1440 píxeles de alto y ancho.
- **4K**: 3840 x 2160 píxeles de alto y ancho.
- **5K**: 5120 x 2160 píxeles de alto y ancho.
- **8K**: 7680 x 4320 píxeles.

Primero vamos a analizar los tiempos de ejecución en detalle del algoritmo en el clúster configurado con 1, 2, 3 y 4 nodos. La Figura 13 muestra los tiempos de: dividir la imagen de entrada, copiar los datos (programa, *kernel,* fragmento de la imagen i-ésimo) desde el nodo maestro al nodo i-ésimo, ejecutar la transformación, copiar los resultados desde el nodo i al nodo maestro, juntar los resultados y de otras operaciones (esperar a que todos los nodos finalicen, realizar la actualización del fichero de control, etc). Se obtienen las siguientes conclusiones:

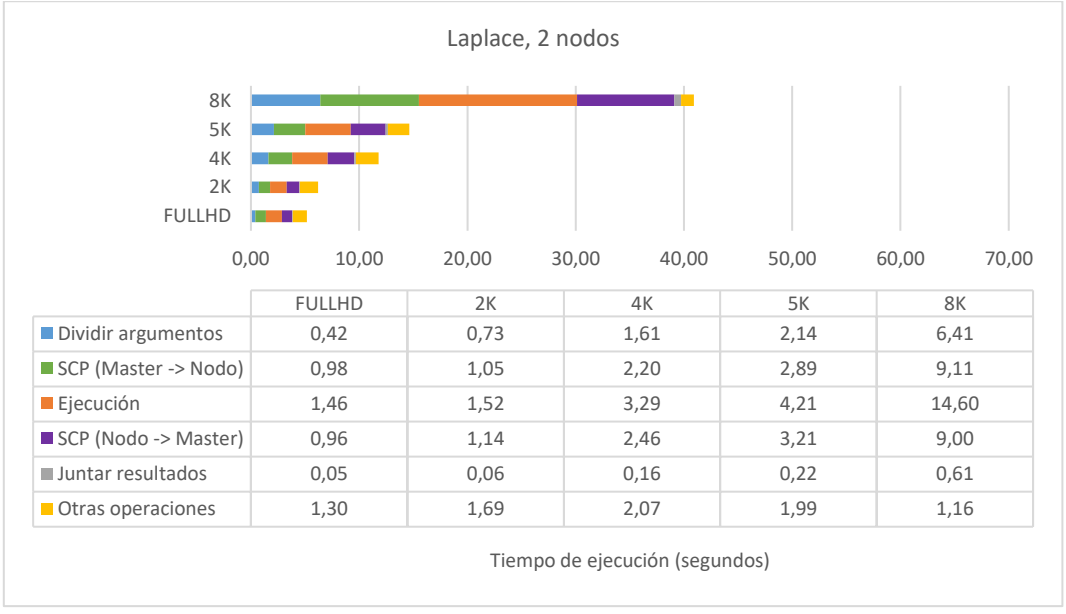
- Antes de nada, notaremos que el tiempo de **dividir los argumentos** y de juntar los resultados es más o menos el mismo en el caso de tener 1 nodo ó 2, 3, 4. Esto se debe a que, en el código del lanzador, se han simplificado estas tareas para que se hagan siempre. Es decir, si tenemos un nodo no sería necesario ni dividir los argumentos ni juntar los resultados, pero se ha optado por hacerlo igualmente. En los casos en los que se tiene más de un nodo, se ha tomado como referencia el tiempo más largo.
- Lo primero que notamos es que el **tiempo de ejecución** (representado en color naranja) aumenta a medida que incrementamos la resolución de la imagen de entrada, en el caso de 1 nodo. Al aumentar el número de nodos esta diferencia disminuye, hasta ser prácticamente igual.

- El tiempo empleado en **copiar** los datos del nodo maestro a los nodos trabajadores (color verde) es siempre ligeramente superior al tiempo empleado en copiar los resultados de los trabajadores al maestro (color morado); con la excepción de la imagen 8K en el caso de 4 nodos. En este último caso puede deberse a un estado de saturación de la red.
- El tiempo de "Otras operaciones" aumenta al aumentar el número de nodos. Esto es normal puesto que, como recordaremos, este tiempo es la suma de los tiempos de esperar a que todos los nodos trabajadores finalicen y de actualizar los ficheros de estado.
- Otra **conclusión** interesante es que dividir la imagen de entrada en fragmentos tiene mayor coste que juntar los resultados. Por ejemplo, en el caso de dividir la imagen 4K en el caso de 3 nodos, se emplean 1,64 segundos mientras que en juntar los 3 resultados se tarda 0,16 segundos. Casi 10 veces menos.



	FULLHD	2K	4K	5K	8K
Dividir argumentos	0,41	0,71	1,58	2,13	6,39
SCP (Master -> Nodo)	1,51	1,88	4,03	5,20	15,51
Ejecución	1,92	2,49	6,04	7,79	25,73
SCP (Nodo -> Master)	1,53	1,91	4,54	5,95	15,52
Juntar resultados	0,05	0,07	0,17	0,27	0,64
Otras operaciones	0,72	0,75	0,78	0,69	1,12

Tiempo de ejecución (segundos)



	FULLHD	2K	4K	5K	8K
Dividir argumentos	0,42	0,73	1,61	2,14	6,41
SCP (Master -> Nodo)	0,98	1,05	2,20	2,89	9,11
Ejecución	1,46	1,52	3,29	4,21	14,60
SCP (Nodo -> Master)	0,96	1,14	2,46	3,21	9,00
Juntar resultados	0,05	0,06	0,16	0,22	0,61
Otras operaciones	1,30	1,69	2,07	1,99	1,16

Tiempo de ejecución (segundos)

	FULLHD	2K	4K	5K	8K
Dividir argumentos	0,41	0,72	1,64	2,19	6,42
SCP (Master -> Nodo)	1,47	1,85	3,05	3,77	9,49
Ejecución	0,75	0,98	1,62	2,82	5,07
SCP (Nodo -> Master)	0,57	0,75	1,22	2,36	6,61
Juntar resultados	0,05	0,01	0,16	0,22	0,63
Otras operaciones	1,91	2,27	3,17	1,63	4,30

	FULLHD	2K	4K	5K	8K
Dividir argumentos	0,41	0,72	1,68	2,18	6,37
SCP (Master -> Nodo)	1,70	2,12	3,16	3,99	4,72
Ejecución	0,68	0,84	1,33	1,62	3,88
SCP (Nodo -> Master)	0,54	0,90	1,47	2,61	6,62
Juntar resultados	0,05	0,08	0,16	0,22	0,65
Otras operaciones	2,00	1,76	2,42	1,34	7,25

Figura 13: Ejecución detallada del algoritmo de Laplace sobre el clúster

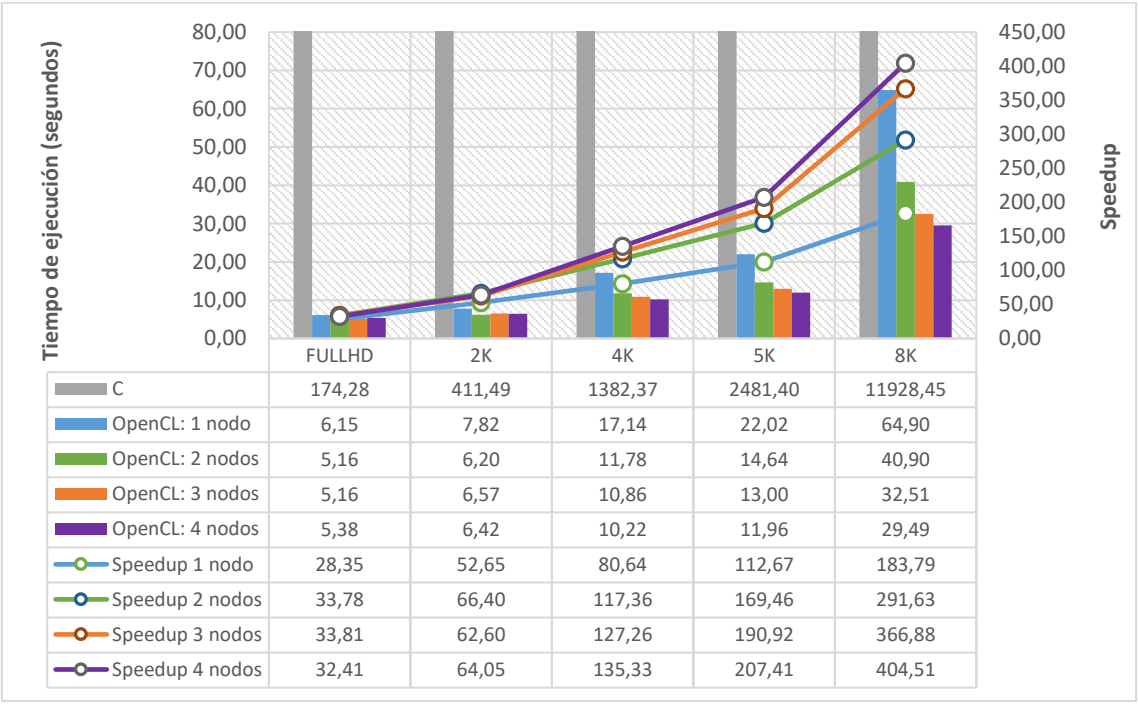
Y, ahora, vamos a analizar los resultados teniendo en cuenta únicamente el tiempo de ejecución total. La Figura 14 ilustra un análisis de escalabilidad mostrando tiempos de ejecución en C y OpenCL. En el caso de C se han ejecutado de manera local, es decir, desde la placa. En el caso de *OpenCL* se han ejecutado en el clúster por medio del lanzador y se ha medido el tiempo desde que se lanza el algoritmo hasta que todos sus nodos finalizan. Podemos observar lo siguiente:

NOTA: los valores del *speedup* se han tomado con respecto a los tiempos de ejecución en C. Esto será así para todos los casos a menos que se diga lo contrario.

- El código en C, que se ejecuta en el procesador *ARM* de la DE1-SOC, se ejecuta varias veces más lento que el código en *OpenCL*, que se ejecuta en la FPGA de la DE1-SOC. En el caso más extremo (imagen 8k), el *speedup* es de 183,79 al ejecutar el programa en

OpenCL sobre el clúster con un nodo frente a hacerlo en C. En casi todos los casos, al aumentar la resolución de la imagen y el número de nodos el *speedup* aumenta.

- Con la imagen *FullHD* apenas hay diferencias entre 1 y 4 nodos. El rendimiento mejora pasando de 1 nodo a 2 o 3 pero, curiosamente, empeora si añadimos un cuarto nodo. Esto se puede deber a que no merece la pena dividir y procesar un fragmento de imagen más pequeño, ya que la latencia de las conexiones es mayor que el tiempo de ejecución.
- Con la imagen 2K obtenemos algo parecido a lo de antes. Si pasamos de 1 nodo a 2 el tiempo de ejecución es menor, pero si añadimos un tercer y cuarto nodo el tiempo es peor.
- Con la imagen 4k la cosa empieza a cambiar: añadir placas reduce el tiempo de ejecución. Pasando de 1 nodo a 2 el tiempo de ejecución se reduce en un 31,27%, mientras que el tiempo de 1 nodo a 3 se reduce en tan sólo un 36,63%. Podemos empezar a pensar que la red de conexión *Fast Ethernet* que hemos utilizado para crear el clúster no es suficiente y es el cuello de botella del sistema.
- Con la imagen 5k obtenemos las mismas conclusiones que en el caso anterior, solo cambia que el porcentaje de mejora al añadir nodos es ligeramente superior.
- Con la imagen 8K alcanzamos el tamaño de entrada óptimo para que merezca la pena emplear más nodos en el clúster. Pasando de 1 nodo a 2 el tiempo mejora en un 36,97%. De 1 nodo a 3 el tiempo mejora en un 49,9%. Y pasando de 1 nodo a 4 el tiempo mejora en un 54,56%. Notamos que duplicar el número de nodos no reduce el tiempo a la mitad, como podríamos pensar. Ha sido necesario cuadruplicar el número de nodos para reducir el tiempo de ejecución en *OpenCL* a la mitad. Esto puede deberse a la complejidad del algoritmo, que obliga a aumentar 4 veces el número de unidades de cómputo para obtener un *speedup* igual o superior a 1,5.



	FULLHD	2K	4K	5K	8K
C	174,28	411,49	1382,37	2481,40	11928,45
OpenCL: 1 nodo	6,15	7,82	17,14	22,02	64,90
OpenCL: 2 nodos	5,16	6,20	11,78	14,64	40,90
OpenCL: 3 nodos	5,16	6,57	10,86	13,00	32,51
OpenCL: 4 nodos	5,38	6,42	10,22	11,96	29,49
Speedup 1 nodo	28,35	52,65	80,64	112,67	183,79
Speedup 2 nodos	33,78	66,40	117,36	169,46	291,63
Speedup 3 nodos	33,81	62,60	127,26	190,92	366,88
Speedup 4 nodos	32,41	64,05	135,33	207,41	404,51

Figura 14: Comparativa de la ejecución del algoritmo de Laplace con 1 a 4 nodos

4.4 Experimento II: Detección de bordes con el operador de Sobel

Este algoritmo se parece bastante al anterior, pero hemos aumentado un poco la complejidad aprovechando que hace uso de 2 matrices de coeficientes; no sólo una como en el caso de *Laplace*. Al igual que antes, partimos del código de ejemplo "*Sobel Filter Design Example*" y lo modificamos para evitar que utilice *OpenGL* y la librería *SDL* para mostrar el resultado del filtro en tiempo real. El algoritmo aplicará el filtro con 2 matrices de coeficientes sobre una imagen pasada como parámetro y finalizará.

Como acabamos de comentar, este algoritmo hace uso de 2 matrices de coeficientes. Una para las componentes horizontales y otra para las verticales. También se basa en el teorema de la diferencia central, pero dando mayor peso a los píxeles centrales. Para hacer esto, basta con dejar a 0 la línea central de la matriz:

$$Gx = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix},\ para\ las\ componentes\ x\ (horizontales)$$

$$Gy = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix},\ para\ las\ componentes\ y\ (verticales)$$

La entrada es una imagen en color en RGB, con una profundidad de 24 bits y de formato PPM. La salida será también una imagen en formato PPM, pero será bicolor: blanco para los bordes y negro para el resto. La Figura 15 ilustra el resultado de aplicar el filtro con el operador de Laplace (imagen inferior) sobre una imagen de 1920 x 1080 píxeles (arriba):

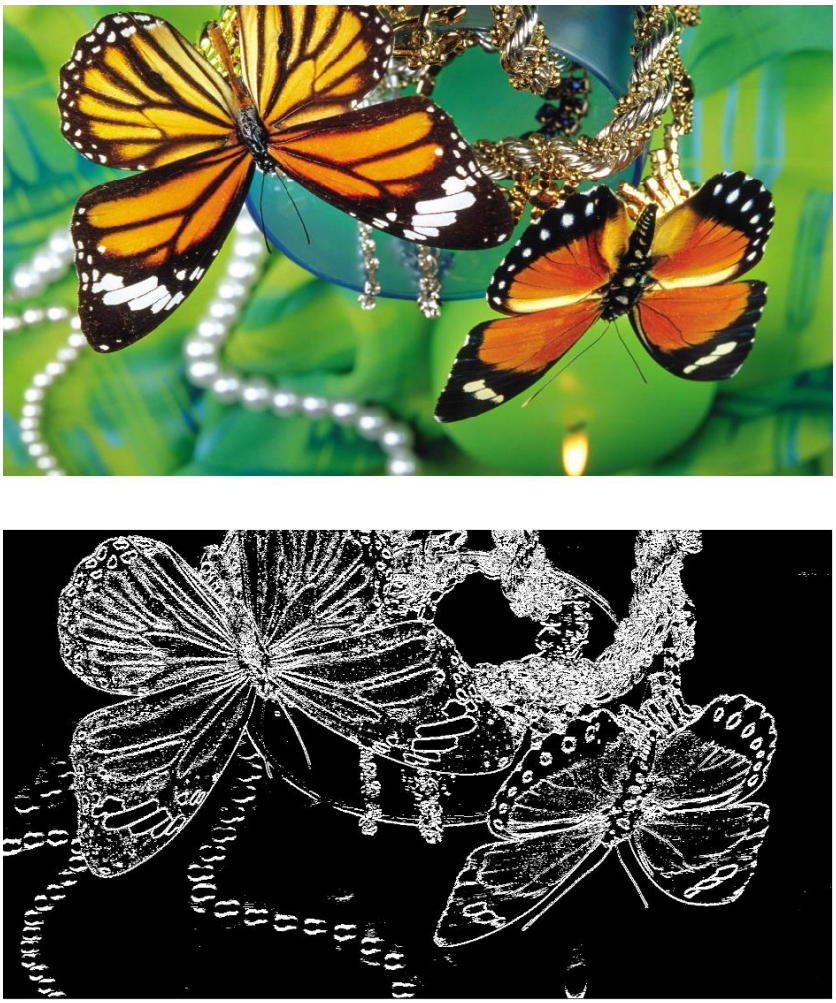


Figura 15: Imagen de resultado detección de bordes de Sobel

4.4.1 Código del kernel de *OpenCL*

El código del programa que se ejecutará en la *FPGA* es muy similar al del algoritmo de *Laplace*. La principal diferencia reside en el bucle que aplica las matrices de los coeficientes. En este caso y con el fin de aumentar ligeramente la complejidad respecto al algoritmo anterior, se ha dividido en 2: uno para las componentes horizontales y otro para las verticales. Como se puede observar, el bucle de las horizontales utiliza la conversión de los píxeles de RGB a LUMA igual que antes; aproximando con operaciones de enteros:

```
__kernel void sobel(
     global unsigned int * restrict frame_in,
     global unsigned int * restrict frame_out,
     const int iterations,
     const unsigned int threshold) {

    // Filter coefficients
    int Gx[3][3] = {{-1,-2,-1},{0,0,0},{1,2,1}};
    int Gy[3][3] = {{-1,0,1},{-2,0,-2},{-1,0,1}};
```

```
// Pixel buffer of 2 rows and 3 extra pixels
int rows[2 * COLS + 3];
// The initial iterations are used to initialize the pixel buffer.
int count = -(2 * COLS + 3);

while (count != iterations) {

. . .

    int x_dir = 0;
    #pragma unroll
    for (int i = 0; i < 3; ++i) {
        #pragma unroll
        for (int j = 0; j < 3; ++j) {
          unsigned int pixel = rows[i * COLS + j];
          unsigned int b = pixel & 0xff;
          unsigned int g = (pixel >> 8) & 0xff;
          unsigned int r = (pixel >> 16) & 0xff;

          // RGB -> Luma conversion (integer approximation)
          unsigned int luma = r * 66 + g * 129 + b * 25;
          luma = (luma + 128) >> 8;
          luma += 16;

          x_dir += luma * Gx[i][j];
        }
    }
```

Sin embargo, las componentes verticales hacen uso de una conversión mediante una aproximación de números reales:

```
    int y_dir = 0;

    #pragma unroll
    for (int i = 0; i < 3; ++i) {
        #pragma unroll
        for (int j = 0; j < 3; ++j) {
            unsigned int pixel = rows[i * COLS + j];
            unsigned int b = pixel & 0xff;
            unsigned int g = (pixel >> 8) & 0xff;
            unsigned int r = (pixel >> 16) & 0xff;

            // RGB -> Luma conversion (float approximation)
            float luma_aux =
                   sqrt(0.299f*r*r + 0.587f*g*g + 0.114f*b*b);
            luma_aux += 0.5;
            unsigned int luma = ( (int)luma_aux + 128) >> 8;
            luma += 16;
```

```
                y_dir += luma * Gy[i][j];
            }
          }
```

Y, al final, juntamos ambas componentes en una sola y comprobamos si están por encima o por debajo del umbral para dibujar el píxel de blanco o negro:

```
        int temp = abs(x_dir) + abs(y_dir);
        unsigned int clamped;
        if (temp > threshold) {
            clamped = 0xffffff;
        } else {
            clamped = 0;
        }

        if (count >= 0) {
            frame_out[count] = clamped;
        }
        count++;
    }
}
```

Con este cambio aumentamos el número de operaciones de enteros y añadimos operaciones en coma flotante que se ejecutan dentro del bucle más interno. Esto provoca un aumento ligero del tiempo de ejecución. Se llegan a realizar WIDTH * HEIGHT * 3 * 3 * 10 (6 productos, 3 sumas y una raíz cuadrada) operaciones en coma flotante extra respecto a la implementación de *Laplace*.

4.4.2 Lanzador del algoritmo en el *clúster*

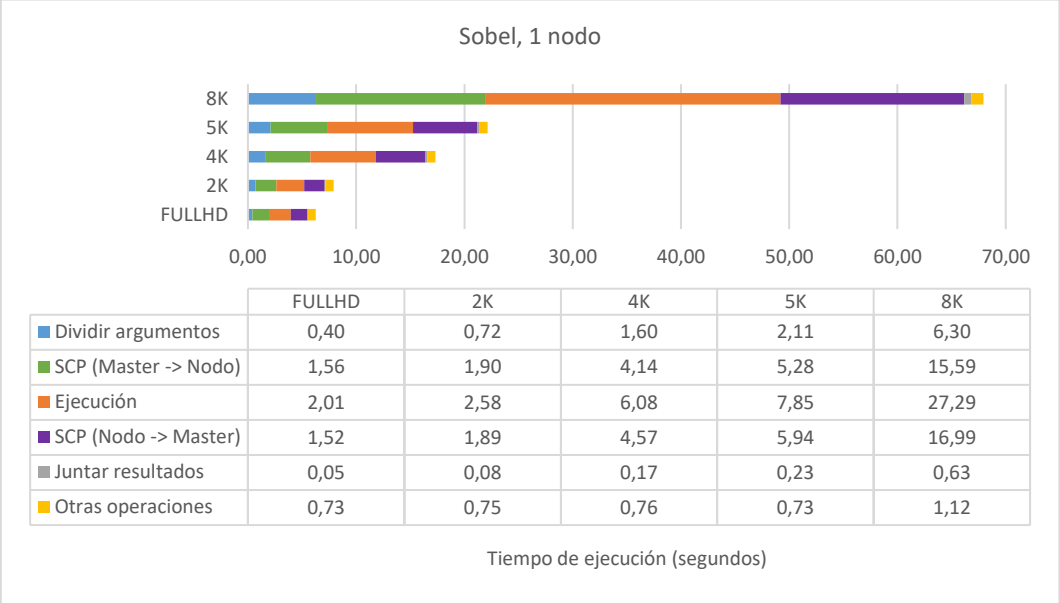
El lanzador es exactamente igual al lanzador de *Laplace*. Lo único que cambia es el valor del umbral que hay que pasarle a la función como parámetro. En el caso de *Laplace* el valor era 32. En *Sobel*, el valor óptimo es de 128. El código completo puede verse en el mismo apartado que antes, el Anexo I.

4.4.3 Resultados

Al igual que en el apartado anterior, vamos a analizar los tiempos de ejecución del algoritmo de *Sobel* sobre el clúster con 1, 2, 3 y 4 nodos y las mismas 5 imágenes en formato PPM. Primero veamos las gráficas y tablas de los tiempos detallados, como nos muestra la Figura 16. Apenas hay diferencias en cuanto a las conclusiones respecto al caso de *Laplace*:

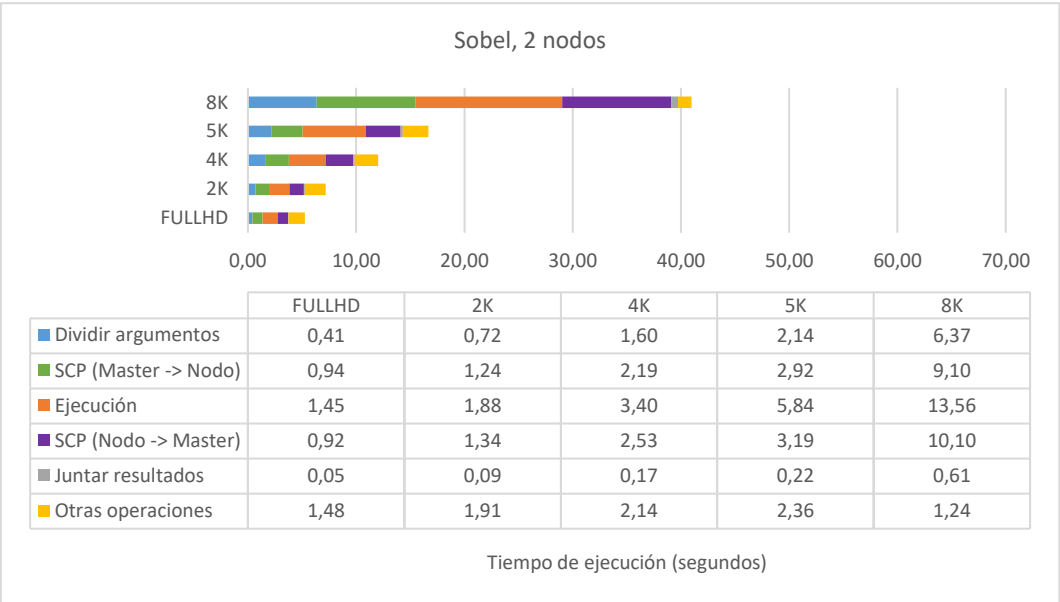
- El tiempo de **ejecución** aumenta al incrementar la resolución de la imagen de entrada, salvo en el caso de ser ejecutado en 4 nodos.

- El tiempo de **copia** del nodo maestro a los nodos esclavos es siempre mayor que en el caso contrario.
- **Dividir** la imagen de entrada es muchísimo más costoso que juntar los resultados.
- La mayor **mejora** se produce con la imagen más grande, dada la complejidad del problema. Para casos de imágenes de baja resolución, tal vez no merezca la pena pasar de 2 nodos. Sin embargo, si el problema fuese procesar un conjunto muy grande N de imágenes de baja resolución, sí merecería la pena aumentar el número de nodos del *clúster*. Para ello sería necesario cambiar la estrategia del lanzador: enviar cada imagen de entrada directamente (sin dividirla) a cada uno de los nodos. En el caso de imágenes 4k, habría que dividirlas en 2, procesarlas en 2 nodos y juntar las 2 imágenes resultantes; y hacer esto en paralelo con bloques de 2 nodos.



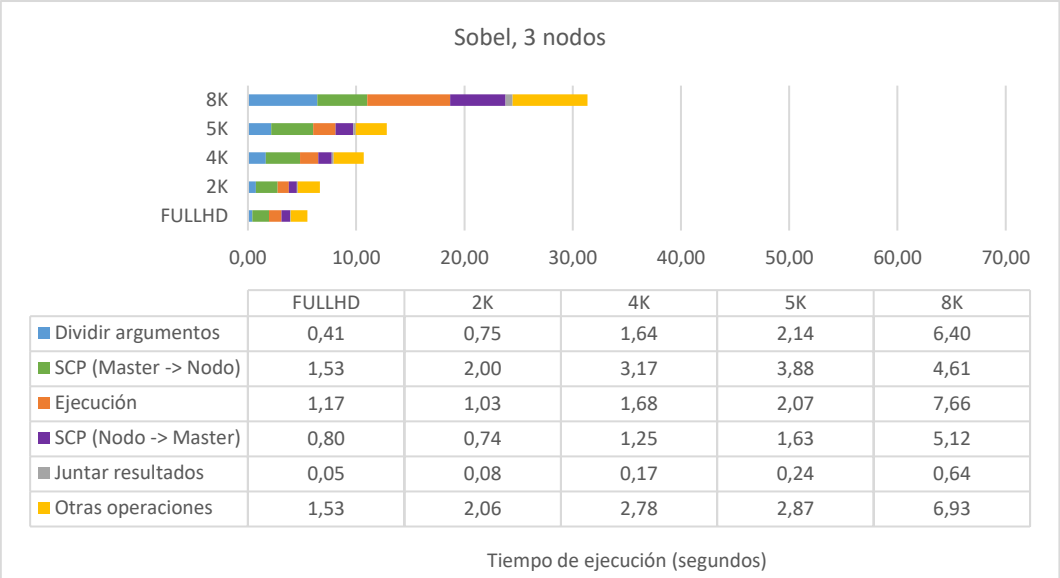
	FULLHD	2K	4K	5K	8K
Dividir argumentos	0,40	0,72	1,60	2,11	6,30
SCP (Master -> Nodo)	1,56	1,90	4,14	5,28	15,59
Ejecución	2,01	2,58	6,08	7,85	27,29
SCP (Nodo -> Master)	1,52	1,89	4,57	5,94	16,99
Juntar resultados	0,05	0,08	0,17	0,23	0,63
Otras operaciones	0,73	0,75	0,76	0,73	1,12

Tiempo de ejecución (segundos)



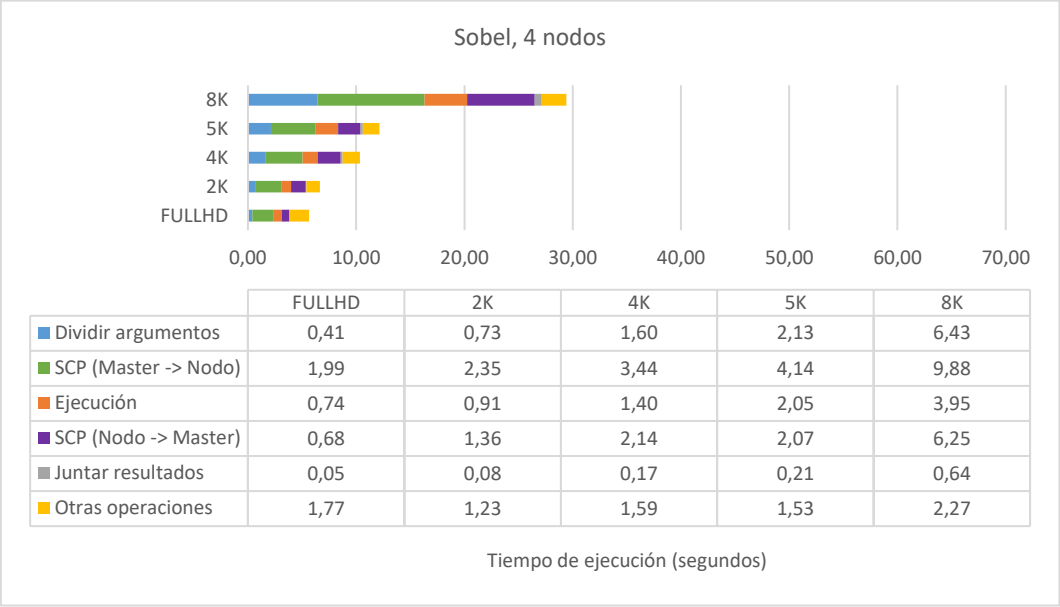
	FULLHD	2K	4K	5K	8K
Dividir argumentos	0,41	0,72	1,60	2,14	6,37
SCP (Master -> Nodo)	0,94	1,24	2,19	2,92	9,10
Ejecución	1,45	1,88	3,40	5,84	13,56
SCP (Nodo -> Master)	0,92	1,34	2,53	3,19	10,10
Juntar resultados	0,05	0,09	0,17	0,22	0,61
Otras operaciones	1,48	1,91	2,14	2,36	1,24

Tiempo de ejecución (segundos)



	FULLHD	2K	4K	5K	8K
Dividir argumentos	0,41	0,75	1,64	2,14	6,40
SCP (Master -> Nodo)	1,53	2,00	3,17	3,88	4,61
Ejecución	1,17	1,03	1,68	2,07	7,66
SCP (Nodo -> Master)	0,80	0,74	1,25	1,63	5,12
Juntar resultados	0,05	0,08	0,17	0,24	0,64
Otras operaciones	1,53	2,06	2,78	2,87	6,93

Tiempo de ejecución (segundos)



	FULLHD	2K	4K	5K	8K
Dividir argumentos	0,41	0,73	1,60	2,13	6,43
SCP (Master -> Nodo)	1,99	2,35	3,44	4,14	9,88
Ejecución	0,74	0,91	1,40	2,05	3,95
SCP (Nodo -> Master)	0,68	1,36	2,14	2,07	6,25
Juntar resultados	0,05	0,08	0,17	0,21	0,64
Otras operaciones	1,77	1,23	1,59	1,53	2,27

Tiempo de ejecución (segundos)

Figura 16: Ejecución detallada del algoritmo de Sobel sobre el clúster

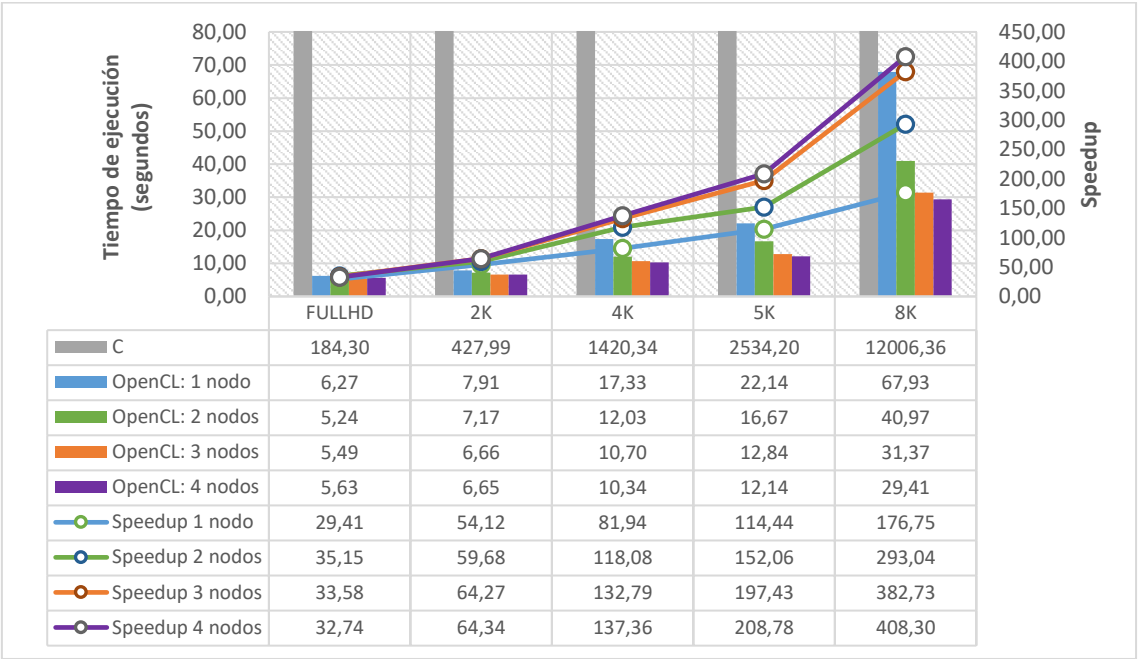
En la Figura 17 se han reunido los tiempos, de manera simplificada, de ejecutar el algoritmo en C y en *OpenCL*. En éste nuevo análisis de escalabilidad, se observa lo siguiente:

- Al igual que antes, los tiempos de ejecución en **C** son mucho más elevados que en el caso de *OpenCL*. Esta diferencia aumenta al aumentar el tamaño de los datos de entrada. El *speedup* vuelve a aumentar a medida que aumenta la resolución de la imagen de entrada y el número de nodos. Observamos cómo esta mejora se hace más holgada en el caso de la imagen 8k al añadir más nodos.

- Apenas mejora el **rendimiento** al utilizar más nodos para procesar las imágenes de resolución *FullHD* y 2K. En la imagen *FullHD* el tiempo mejora al pasar de 1 nodo a 2, pero al añadir 3 y 4, el tiempo empeora ligeramente. En el caso de la imagen 2K el tiempo siempre mejora, aunque cada vez menos. Si observamos los resultados en detalle con

las gráficas anteriores, comprobamos que se debe a que el tiempo de "Otras operaciones" aumenta al aumentar el número de nodos.

- En el resto de las imágenes se produce una gran **reducción** del tiempo de ejecución, pero solamente al pasar de 1 nodo a 2. Hablamos de una mejora de entre el 30% al 39%. Al igual que en el caso de *Laplace* son necesarios 4 nodos para reducir el tiempo de ejecución de 1 nodo por debajo de la mitad. Parece ser que la red de interconexión no proporciona el ancho de banda necesario para que los tiempos de envío de datos sean inferiores a los tiempos de procesamiento.



	FULLHD	2K	4K	5K	8K
C	184,30	427,99	1420,34	2534,20	12006,36
OpenCL: 1 nodo	6,27	7,91	17,33	22,14	67,93
OpenCL: 2 nodos	5,24	7,17	12,03	16,67	40,97
OpenCL: 3 nodos	5,49	6,66	10,70	12,84	31,37
OpenCL: 4 nodos	5,63	6,65	10,34	12,14	29,41
Speedup 1 nodo	29,41	54,12	81,94	114,44	176,75
Speedup 2 nodos	35,15	59,68	118,08	152,06	293,04
Speedup 3 nodos	33,58	64,27	132,79	197,43	382,73
Speedup 4 nodos	32,74	64,34	137,36	208,78	408,30

Figura 17: Comparativa de la ejecución del algoritmo de Sobel con 1 a 4 nodos

4.5 Experimento III: Emborronamiento Gaussiano

El emborronamiento o *blur*[7] es una transformación muy utilizada en el procesamiento de imágenes. El principal objetivo de aplicarla consiste en eliminar el ruido de una imagen, aunque también puede utilizarse para detectar bordes. Por ejemplo, podríamos aplicar el filtro gaussiano a una imagen y, después, aplicar la detección de bordes con los algoritmos de *Sobel* y *Laplace* que acabos de ver para obtener un resultado mucho más definido.

La teoría detrás es muy sencilla: simplemente hay que calcular, por cada píxel, la media con respecto a sus vecinos. La forma más sencilla de conseguir esta transformación es utilizar una matriz de coeficientes, como en los casos anteriores, que aproxime la media sin necesidad de calcularla con los valores de cada píxel. Esta matriz, de tamaño n x m, podría ser algo así:

[7] https://en.wikipedia.org/wiki/Gaussian_blur

$$\frac{1}{n*m}*\begin{pmatrix}1 & \cdots & 1\\ \vdots & \ddots & \vdots\\ 1 & \cdots & 1\end{pmatrix}$$

Sin embargo, hemos optado por una implementación algo más compleja, y diferenciadora, con respecto a las anteriores basada en la distribución Normal o de Gauss de 2 dimensiones:

$$G(x,y) = \frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Donde x es la distancia al origen en el eje horizontal, y es la distancia al origen desde el eje vertical y σ es la desviación de la distribución de *Gauss.* Si observamos la Figura 18, observamos cómo el valor central de x alcanza el máximo valor de y; y cómo, al alejarse x del centro, el valor de y va disminuyendo paulatinamente y de manera constante.

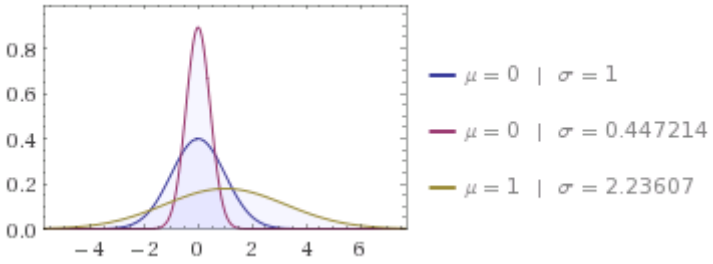


Figura 18: Representación gráfica de la Distribución Normal o de Gauss en 1-D

Para este caso, el valor central de x sería el color de cada uno de los píxeles de la imagen de entrada y los valores que se disminuyen serían los colores de sus vecinos. De esta forma, provocaríamos un gradiente de color que produce el efecto de difuminado. Lo mejor es que, variando los valores de la desviación, el efecto es mayor o menor. La Figura 19 es un ejemplo obtenido de la *Wikipedia*[8].

Figura 19: Efecto de variar la desviación del Emborronamiento Gaussiano

[8] https://en.wikipedia.org/wiki/Gaussian_blur

La entrada del programa será una imagen en formato PPM como las de antes, sin comentarios, formato de color RGB y profundidad de color de 24 bits. También habrá que especificar el alto y el ancho de la imagen y la ruta de destino del resultado, el valor de sigma y el radio. La imagen de salida también tendrá el formato PPM. La Figura 20 muestra el resultado (imagen inferior izquierda) de ejecutar el programa sobre una imagen (imagen superior izquierda), fijando los valores de sigma a 5 y de radio a 10. En la parte derecha se han ampliado cada una de las imágenes para observar mejor este efecto.

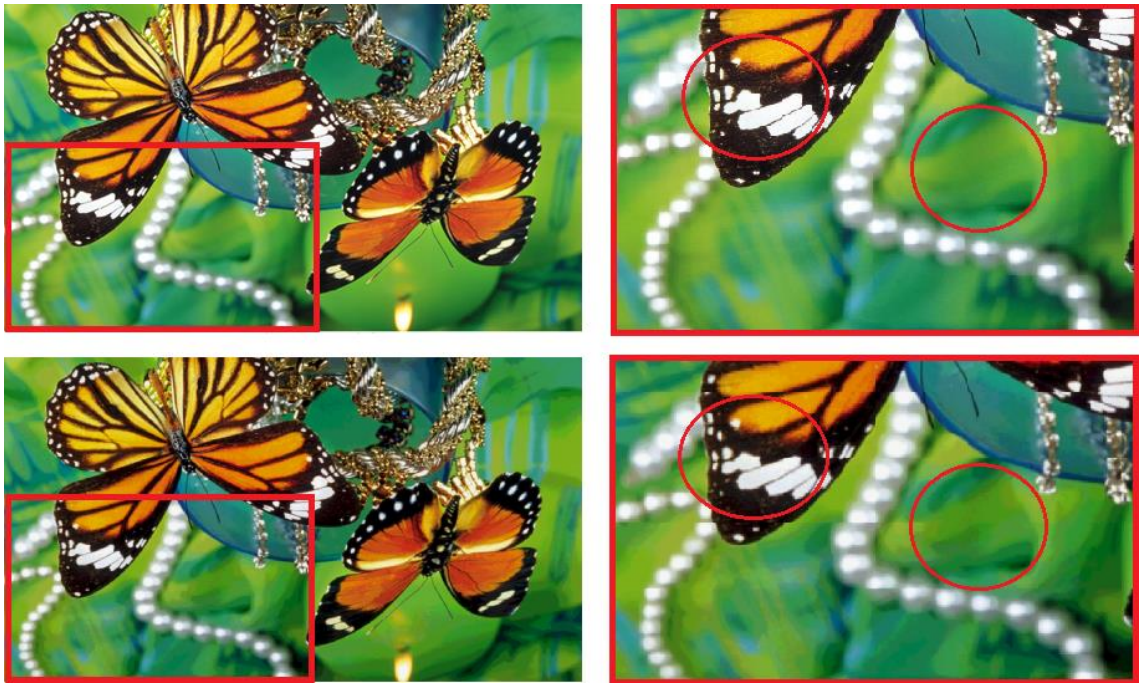


Figura 20: Imagen de resultado del Emborronamiento Gaussiano

4.5.1 Código del kernel de *OpenCL*

Nuestra implementación calculará la matriz de coeficientes según la fórmula de la distribución Normal o de *Gauss* fijada la desviación con el parámetro de entrada *sigma.* Después, la aplicará cada píxel y a sus vecinos en una distancia fija, tomada también como parámetro. El cálculo de la matriz de coeficientes se hará en el programa Main, que se ejecuta en el host. Mientras que la aplicación de la matriz se hará en el programa *kernel* de *OpenCL.*

El cálculo de la matriz de coeficientes de *Gauss* consiste en implementar la fórmula de la distribución normal:

```
double gauss_2d(int x, int y, double sigma) {

   double result = 1.0 / (2 * PI * sigma * sigma);
   result *= exp(-(x*x+y*y)/(2 * sigma * sigma));

   return result;
}
```

Y aplicarla un número *dim* específico de veces. Este número es igual a 2 veces el radio más uno al cuadrado porque, más adelante en el *kernel*, la matriz se aplicará por cada píxel y sus vecinos; tanto los del lado izquierdo, como los del derecho. Una vez hecho esto, se calcula la media y se divide cada valor obtenido al aplicar la distribución por dicha media:

```
double* create_gauss_matrix(int radius, double sigma) {

  //Used for iterations
  int i, j;

  //The matrix width and height
  int dim = 2*radius+1;

  //Alocate mem for the matrix
  double* filter = (double*) malloc(sizeof(double) * dim * dim);

  //Calculate
  double sum = 0.0;
  double g = 0.0;

  for(i = -radius; i <= radius; i++) {
     for(j = -radius; j <= radius; j++) {
        filter[(j+radius) + (i+radius) * dim] = gauss_2d(j, i, sigma);
        sum += filter[ (j+radius) + (i+radius) * dim ];
     }
  }

  //Correct so that the sum of all elements ~= 1
  for(i = 0; i < 2*radius+1; i++) {
     for(j = 0; j < 2*radius+1; j++) {
        filter[j + i * dim] /= sum;
     }
  }

  return filter;
}
```

Por último, sólo nos queda ver el código del *kernel* de *OpenCL*. Definimos un tipo para representar un píxel mediante una estructura y fijamos el valor del radio con una macro de preprocesador. Esto es necesario porque, para desenrollar los bucles y acelerar su ejecución, el compilador tiene que saber en tiempo de compilación los límites de los bucles a desenrollar:

```
#define RADIUS 3

typedef struct {
  unsigned char R;
  unsigned char G;
  unsigned char B;
} pixel;
```

Los parámetros de entrada y salida de esta función son:

- **original (entrada)**: puntero a un array del tipo píxel que contiene la imagen de entrada que se ha leído en el *Main.*
- **filter (entrada):** puntero a la matriz de los coeficientes de *Gauss*, calculada en el *Main.*
- **radius (entrada):** número de píxeles vecinos a los que se aplicará el filtro, por cada píxel.
- **width (entrada):** ancho de la imagen.
- **height (entrada):** alto de la imagen.
- **result (salida):** puntero a un array del tipo píxel que contiene la imagen de resultado.

```
__kernel void blur(
        global pixel* restrict original,
        global pixel* restrict result,
        global double* restrict filter,
        const int radius,
        const int width,
        const int height)
{

  int x, y;
  int dim = 2*RADIUS+1;
  int wr = width-RADIUS;
  int hr = height-RADIUS;
```

El bucle principal recorre cada píxel y los bucles internos aplican la matriz de los coeficientes a cada píxel y a sus vecinos. Los bucles más externos no pueden desenrollarse, puesto que hemos decidido no fijar el tamaño de la imagen. Al contrario de lo que pasaba con los casos de *Sobel y Laplace*, en los que sí decidimos fijarlos:

```
  for(y = 0; y < height; ++y) {
     for(x = 0; x < width; ++x) {
```

De todos modos, aunque se hubiera decidido fijarlos, tampoco podrían ser desenrollados puesto que, a continuación, se hace una comprobación que no puede ser resuelta en tiempo de compilación. Se trata de averiguar si los píxeles están próximos a los bordes. De ser así, no se aplicaría el filtro. Esto es una optimización hecha para poder desenrollar los bucles internos, ya que se evita comprobar dentro de ellos una posible violación de segmento causada por el acceso a una posición de un array fuera de sus límites:

```
        index = x + (y * width);

        if( x < RADIUS || y < RADIUS || x >= wr || y >= hr ) {
                result[index] = original[index];
```

```
            continue;
        }

        int i, j, index=0, index2=0;
        pixel res;
        res.R = res.G = res.B = 0;
        double fil;
```

Estos bucles internos aplican la matriz al píxel de posición (i,j) y a sus vecinos en el radio dado como parámetro. Se hace como una acumulación

```
        #pragma unroll RADIUS
        for(i = -RADIUS; i <= RADIUS; ++i) {

            #pragma unroll RADIUS
            for(j = -RADIUS; j <= RADIUS; ++j) {
                fil = filter[(j+RADIUS) + (i+RADIUS) * dim];
                aux=(x+j) + (y+j) * width;
                res.R += fil * original[aux].R;
                res.G += fil * original[aux].G;
                res.B += fil * original[aux].B;
            }

            result[index].R = res.R;
            result[index].G = res.G;
            result[index].B = res.B;
        }
    }
  }
}
```

4.5.2 Lanzador del algortimo en el *clúster*

Una vez más, el lanzador es el mismo que en los casos de *Laplace* y *Sobel*. Cambian los argumentos finales del programa. En los casos anteriores teníamos que invocar el programa especificando el valor del umbral (*threshsold*). En este caso, tenemos que pasar como argumentos los valores de sigma y el radio y la carpeta de destino de la solución. El código completo puede verse en el Anexo I.

La Figura 21 es una captura de pantalla realizada desde el nodo maestro, en la que se observa la ejecución de este programa con una imagen de 2560x1440 píxeles en el clúster configurado con 2 nodos.

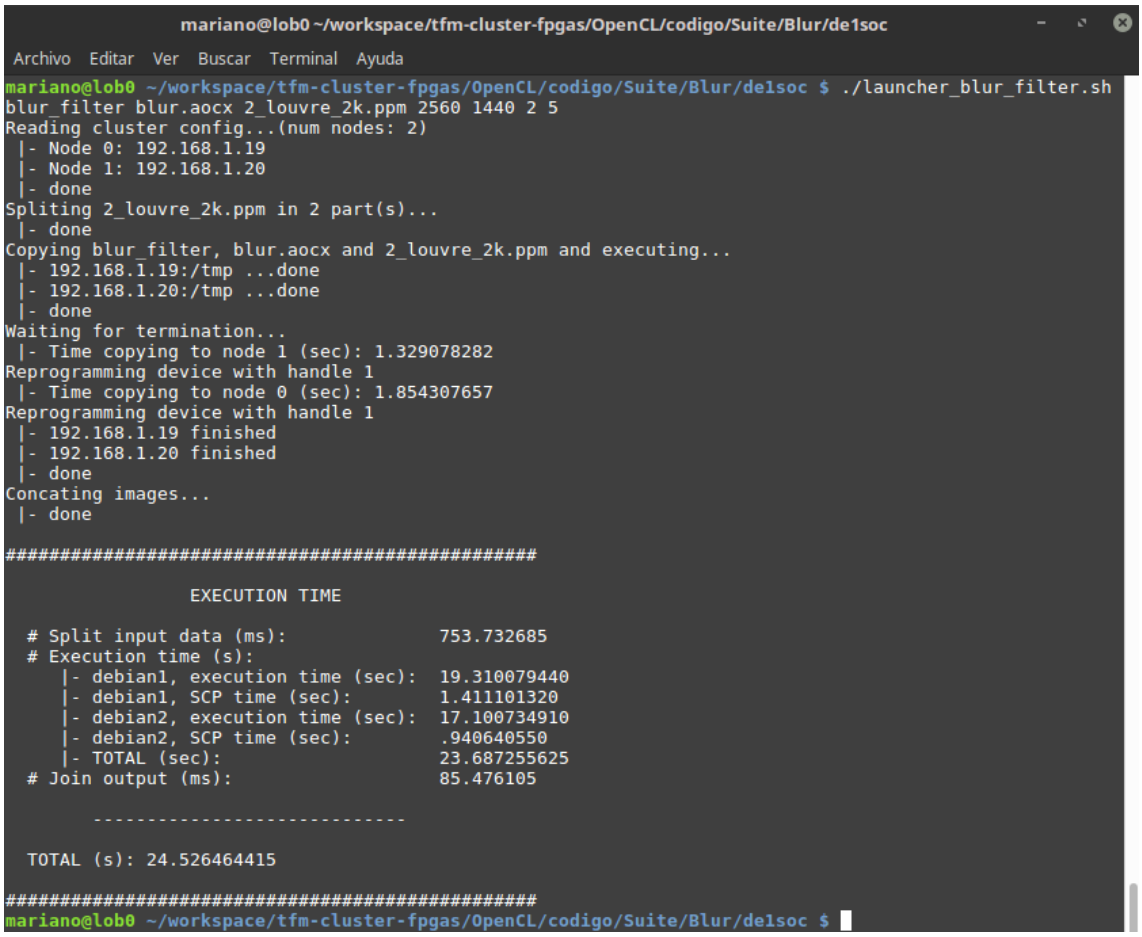
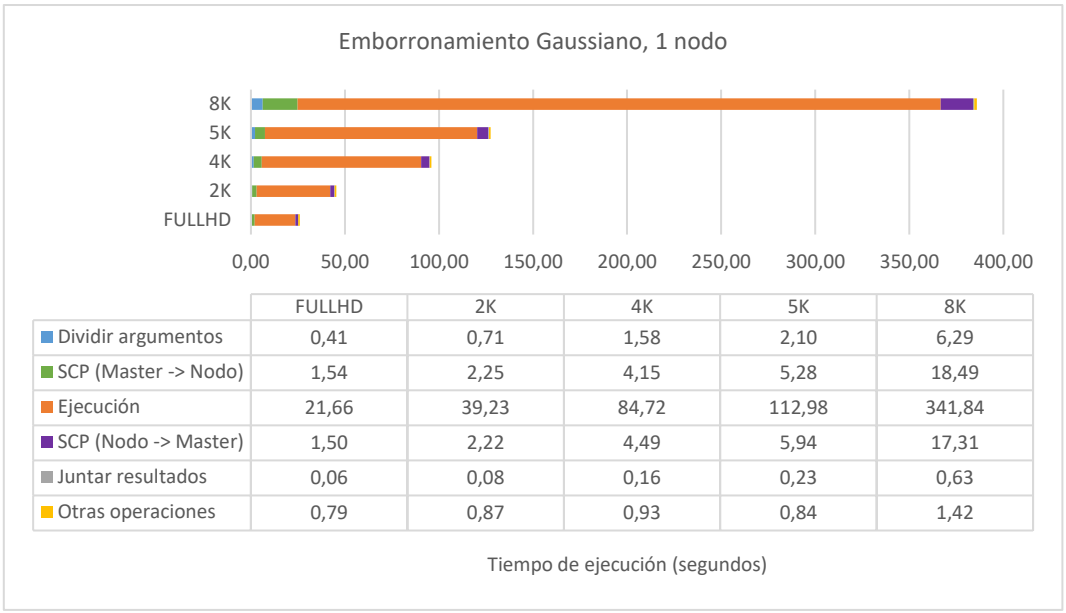


Figura 21: Ejecución del Emborronamiento Gaussiano en el clúster con 2 nodos

4.5.3 Resultados

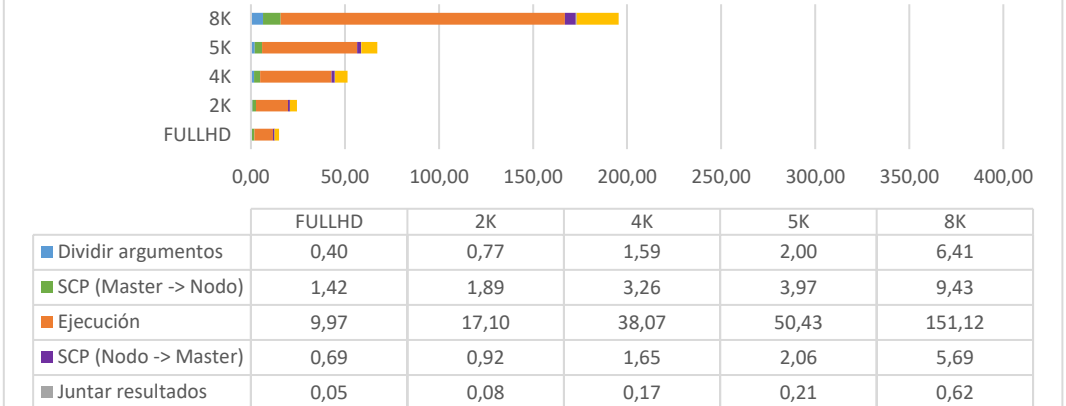
La Figura 22 contiene las gráficas que representan los tiempos de ejecución detallados del algoritmo del Emborronamiento Gaussiano. En este caso, y como se puede apreciar, la mayor parte del tiempo se concentra en la ejecución del programa. Como vimos en el sub apartado anterior, este programa hace uso de una estructura de datos llamada píxel para representar las componentes de color de la imagen de entrada. Esto hace que los cálculos que se realizan sean sobre punteros, y esto repercute enormemente en el rendimiento. Las FPGAs de la DE1-SOC parecen no funcionar bien si el código que ejecutan tiene muchos accesos a memoria.

Al igual que en los casos anteriores, el tiempo de “Otras operaciones” (en color amarillo) aumenta al aumentar el número de nodos ya que este tiempo contiene, entre otras cosas, el tiempo de finalización de los nodos esclavos. Una vez más, el tiempo se reduce considerablemente al pasar de 1 nodo a 2, pero esta diferencia se hace menos notoria al utilizar 3 y 4 nodos. La red vuelve a ser el causante. También observamos que el tiempo de copiar los datos del nodo maestro a los esclavos es siempre mayor al tiempo de hacer esta operación a la inversa.



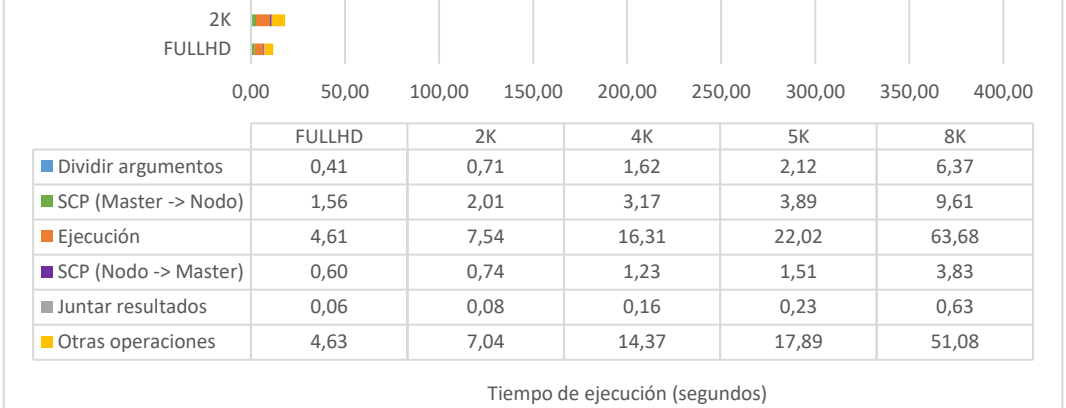
	FULLHD	2K	4K	5K	8K
Dividir argumentos	0,41	0,71	1,58	2,10	6,29
SCP (Master -> Nodo)	1,54	2,25	4,15	5,28	18,49
Ejecución	21,66	39,23	84,72	112,98	341,84
SCP (Nodo -> Master)	1,50	2,22	4,49	5,94	17,31
Juntar resultados	0,06	0,08	0,16	0,23	0,63
Otras operaciones	0,79	0,87	0,93	0,84	1,42

Tiempo de ejecución (segundos)



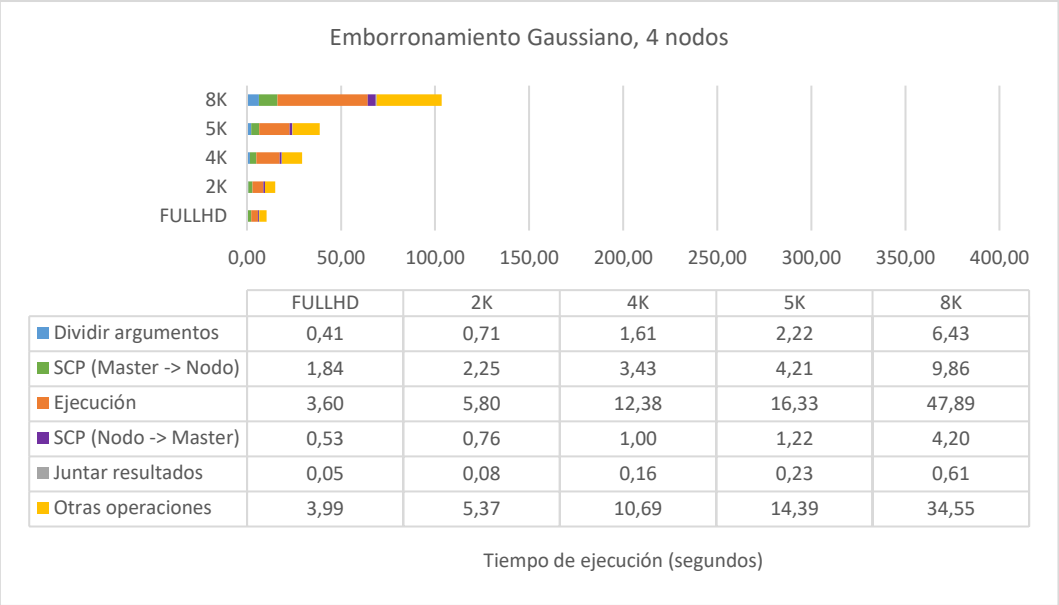
	FULLHD	2K	4K	5K	8K
Dividir argumentos	0,40	0,77	1,59	2,00	6,41
SCP (Master -> Nodo)	1,42	1,89	3,26	3,97	9,43
Ejecución	9,97	17,10	38,07	50,43	151,12
SCP (Nodo -> Master)	0,69	0,92	1,65	2,06	5,69
Juntar resultados	0,05	0,08	0,17	0,21	0,62
Otras operaciones	2,46	3,75	6,70	8,50	22,29

Tiempo de ejecución (segundos)



	FULLHD	2K	4K	5K	8K
Dividir argumentos	0,41	0,71	1,62	2,12	6,37
SCP (Master -> Nodo)	1,56	2,01	3,17	3,89	9,61
Ejecución	4,61	7,54	16,31	22,02	63,68
SCP (Nodo -> Master)	0,60	0,74	1,23	1,51	3,83
Juntar resultados	0,06	0,08	0,16	0,23	0,63
Otras operaciones	4,63	7,04	14,37	17,89	51,08

Tiempo de ejecución (segundos)



	FULLHD	2K	4K	5K	8K
Dividir argumentos	0,41	0,71	1,61	2,22	6,43
SCP (Master -> Nodo)	1,84	2,25	3,43	4,21	9,86
Ejecución	3,60	5,80	12,38	16,33	47,89
SCP (Nodo -> Master)	0,53	0,76	1,00	1,22	4,20
Juntar resultados	0,05	0,08	0,16	0,23	0,61
Otras operaciones	3,99	5,37	10,69	14,39	34,55

Figura 22: Ejecución detallada del algoritmo del Emborronamiento Gaussiano sobre el clúster

La Figura 23 representa los tiempos de ejecución sin detalles, tanto en C como en *OpenCL*. Se han observado las siguientes deducciones:

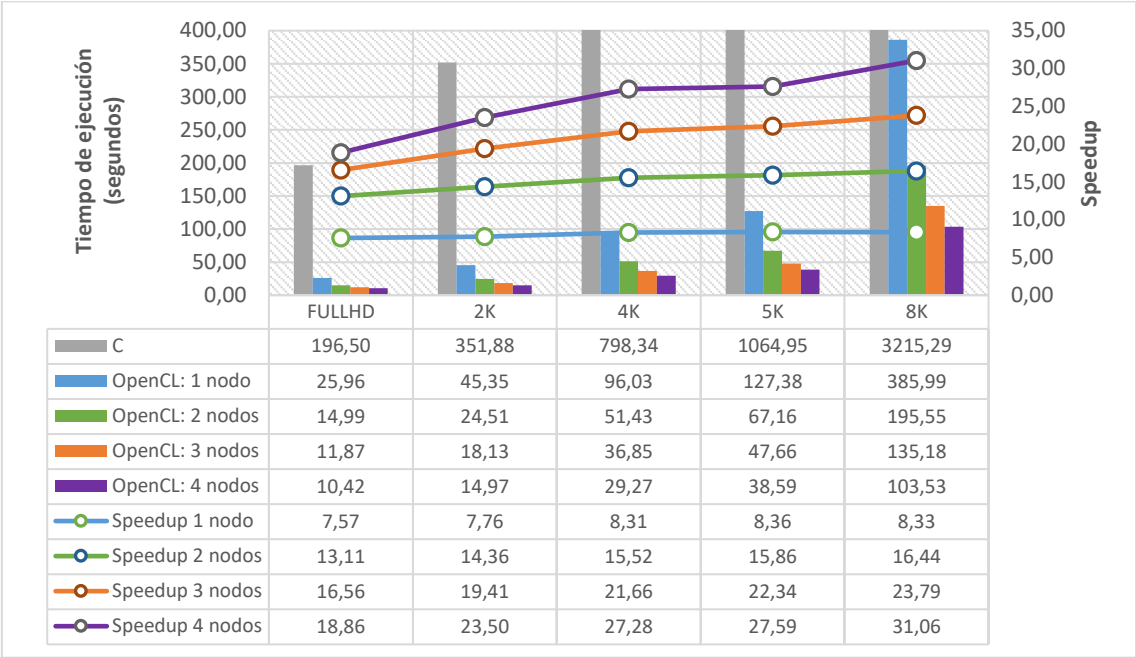
- En todos los casos, el tiempo de ejecución en **C** es muy superior al tiempo en *OpenCL*. En C, el tiempo sigue un crecimiento lineal: el tiempo de pasar de la imagen *FullHD* (1920x1080 píxeles*)* a la 4K (3840x2160 píxeles, cuatro veces una *FullHD*) se multiplica por 4, y pasar de la *FullHD* a la 8k (7680x4320 píxeles, dieciséis veces una *FullHD*), lo hace por 16. Esto no se da en el caso de *OpenCL*: el crecimiento es algo menor al lineal. Y, al aumentar el número de nodos, se reduce aún más.

- El ***speedup*** obtenido al pasar de C a *OpenCL*, en el caso de un solo nodo, oscila entre 7 y 8. Tenemos que tener en cuenta que la ejecución en C se ha hecho de forma local, mientras que la ejecución en *OpenCL* se ha hecho por medio del *script* del programa lanzador. Es decir, en este último caso hay penalización en el tiempo debido a la división de los parámetros de entrada, el retardo de la red, la copia de los resultados, etc. Pero no se pueden ocultar dichos tiempos, puesto que son una de las desventajas del uso de clústeres. A diferencia de en los programas anteriores, las funciones que definen la variación de los *speedup* son bastante constantes.

- El *speedup* más **grande** se obtiene al pasar de 1 nodo a 2. Se aprovecha al máximo el ancho de banda de la red y no se producen cuellos de botella en ella que afecten al rendimiento general del sistema. En el caso de la imagen *FullHD*, la ganancia se acerca a 1,8 y va aumentando a medida que aumenta el tamaño de imagen. Nunca alcanza el valor de 2, lo que supondría reducir el tiempo de ejecución a la mitad, pero es posible que con imágenes de mayor resolución esto sea posible.

- Para los casos de imágenes de resolución menor a 5120x2160 píxeles (5k), la ganancia obtenida al pasar de 3 a 4 nodos en el caso de ***OpenCL*** es inferior a 1,25. Es decir, el tiempo se reduce menos de un 25% al añadir un cuarto nodo. En el caso de la imagen 8k la ganancia obtenida es de 1,31, es decir, el programa se ejecuta un 31% más deprisa.

Por lo tanto, en cuanto a la relación coste/rendimiento, la opción más interesante en *OpenCL* es la de utilizar 3 nodos.



	FULLHD	2K	4K	5K	8K
C	196,50	351,88	798,34	1064,95	3215,29
OpenCL: 1 nodo	25,96	45,35	96,03	127,38	385,99
OpenCL: 2 nodos	14,99	24,51	51,43	67,16	195,55
OpenCL: 3 nodos	11,87	18,13	36,85	47,66	135,18
OpenCL: 4 nodos	10,42	14,97	29,27	38,59	103,53
Speedup 1 nodo	7,57	7,76	8,31	8,36	8,33
Speedup 2 nodos	13,11	14,36	15,52	15,86	16,44
Speedup 3 nodos	16,56	19,41	21,66	22,34	23,79
Speedup 4 nodos	18,86	23,50	27,28	27,59	31,06

Figura 23: Comparativa de la ejecución del algoritmo del Emborronamiento Gaussiano con 1 a 4 nodos

4.6 Experimento IV: El algoritmo del Flujo óptico (Optical flow algorithm)

El problema del flujo óptico [24] consiste en detectar la variación de la posición de los elementos en una escena, típicamente una serie de fotogramas, transcurrida un pequeño período de tiempo; con el fin de calcular los patrones de movimiento de los objetos que la componen.

Para desarrollar este programa, partimos del código de ejemplo "*Optical Flow Desing Example*" que se puede obtener de forma gratuita desde la web de Altera [25]. Implementa el algoritmo del flujo óptico de Lucas-Kanade [26]. Está pensado para ser ejecutado en *FPGAs* con bajos recursos *hardware*, como es el caso de la *Cyclone V* de nuestras placas DE1-SOC. El programa toma como entrada dos imágenes de 1280 x 800 píxeles en formato RAW y genera una imagen en color en formato PPM. Esta imagen de resultado muestra el movimiento de cada uno de los píxeles para cada par de fotogramas de la secuencia inicial mediante el uso de una rueda de colores, como muestra la Figura 24.

Figura 24: Colores según la dirección del algoritmo del flujo óptico

Como se distingue en la Figura 25, los elementos que se han desplazado a la izquierda aparecen en la imagen resultado de color azul, mientras que los que se han desplazado a la derecha lo hacen con el color rojo[9].

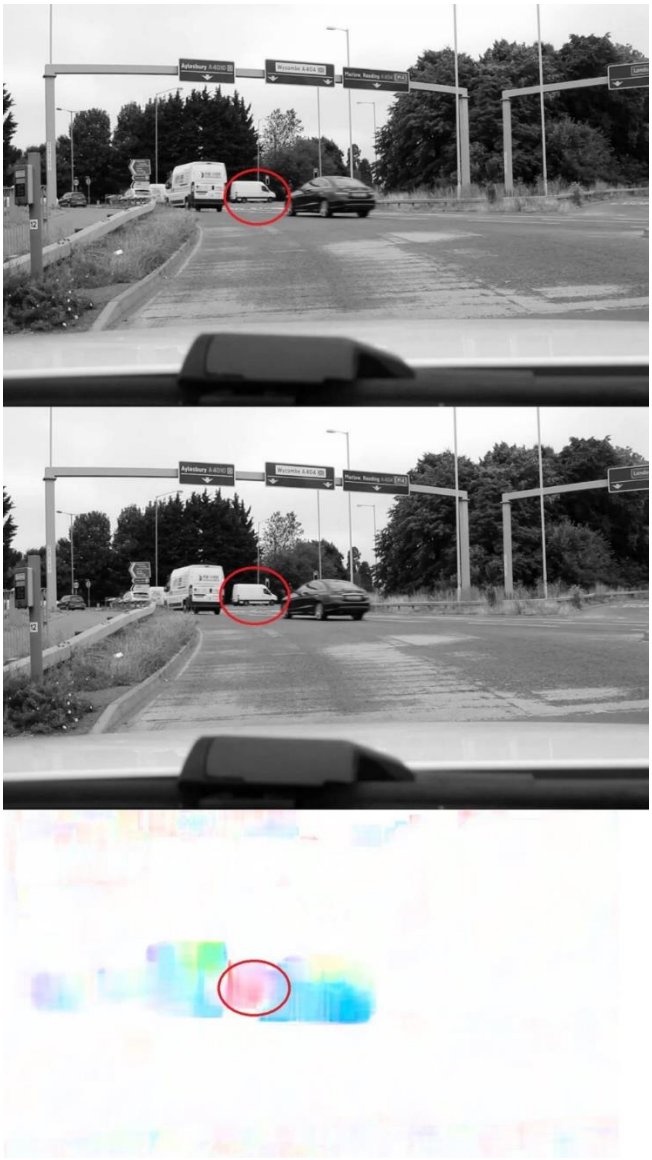


Figura 25: Imagen de resultado flujo óptico

[9] Los círculos de contorno rojo no forman parte de las imágenes procesadas, ni del resultado

Si se amplía la imagen se puede ver con mayor facilidad, en la Figura 26, cómo la furgoneta que hay entre la furgoneta blanca y el coche oscuro, se desplaza hacia la derecha:

Figura 26: Imagen (ampliada) resultado flujo óptico

4.6.1 Código del kernel de *OpenCL*

El algoritmo utiliza el patrón de registro de desplazamiento, o "*shift register pattern*", para reducir al máximo el consumo de memoria, dado que en estas FPGAs es muy limitado. Calcula la variación del movimiento de cada píxel tomando un radio de 56 vecinos adyacentes. Lo que nos da un tamaño de ventana de 7x8 píxeles. Para cada píxel se deben calcular:

- La coordenada x: Lx.
- La coordenada y: Ly.
- Los productos: Lx*Lx, Lx*Ly y Ly*Ly.
- El producto de las coordenadas x e y sobre la longitud del lado(ΔL): ΔL*Lx y ΔL*Ly.

Como se puede observar en el documento llamado *Lucas Kanade Optical Flow – from C to OpenCL on CV SoC de Dmitry Denisenko* [27], este algoritmo ha sufrido cuatro optimizaciones especiales para ser ejecutado en placas con el SoC *Cyclone V.* Nosotros nos centraremos en explicar la versión final. La imagen de entrada se lee en registros de desplazamiento con el fin de optimizar el consumo en memoria. Para cada píxel, en lugar de calcular todas las variables anteriores (Lx, Ly, Lx*Lx…), almacenarlas en registros de desplazamiento y repetir esto 7x8 veces (el tamaño de la ventana); estos valores se calculan al vuelo sobre los registros de desplazamiento que contienen la imagen de entrada. Después se calcula la suma, por columnas, de las variables anteriores y se calcula su cuadrado. De esta forma, reducimos el número de sumas necesarias de 245 a 20 y el número de multiplicaciones de 5 a 10.

El código del *kernel* de *OpenCL* resulta demasiado largo y tedioso como para formar parte de este documento. Por lo tanto, si se desea obtener más información, se requiere ir al enlace de la web de Altera donde se puede descargar de forma gratuita el proyecto preparado para ser ejecutado en sistemas Windows y UNIX/Linux y *FPGAs* de Altera [25].

4.6.2 Lanzador del algoritmo en el *clúster*

En este caso no vamos a dividir las imágenes de entrada en trozos iguales al número de nodos del *clúster*, como hicimos en los casos anteriores. Sino que hemos pensado paralelizarlo mediante un sistema de encuesta. Suponemos que queremos procesar un número N de imágenes. Todas ellas tendrán el mismo prefijo seguido de un número que indica su posición. Por ejemplo, para 10 imágenes podría ser input1.ppm, input2.ppm…input10.ppm. Como el algoritmo toma las imágenes de 2 en 2, al final se generarán N-1 imágenes de resultado. El programa lanzador que se ejecuta en el nodo maestro preguntará a las placas, de una en una en *Round Robin*, por su estado. Si la placa i-ésima está en el estado *idle*, entonces se le copiarán el programa, el *kernel* y las 2 imágenes a procesar y se mandará a ejecutar en segundo plano. Acto seguido se pasará a preguntar al nodo siguiente y así hasta que se procesen todas las imágenes.

El pseudocódigo correspondiente es el siguiente:

```
Parámetros de entrada:
      1. Ejecutable del programa. Extensión ELF.
      2. Kernel de OpenCL compilado. Extension AOCX
      3. Prefijo del nombre de las imágenes (p.e. input, frame)
      4. Número de imágenes
Inicio
      CONFIG=”de1soc-cluster.conf”
      RUNPATH=”/tmp”
      # Directorio donde están las imágenes d entrada
      INPUTPATH=”input/”
      EXT=”.raw”

      Leer las direcciones IP de las placas del fichero CONFIG
      NUM_NODES=número de líneas leídas de CONFIG

      Tiempo0=tomar_tiempo()

      Para i=0 hasta N-1 hacer:
            Para j=0 hasta NUM_NODES hacer:
                  Preguntar por el estado del nodo-j
                  Si el estado == idle entonces
                        Copiar el ejecutable del programa, el fichero del
                        kernel AOCX y las imágenes i e i+1

                        Cambiar el estado del nodo-j por “running”

                        Mandar ejecutar el programa en remoto al
                        nodo-j
                  Sino
                        j = (j+1) módulo NUM_NODES
                  Fin
            Fin
      Fin

      Esperar a que todos los nodos finalicen
      Tiempo2=tomar_tiempo()

      Tiempo_ejecutar=Tiempo1-Tiempo0
      Mostrar tiempos
Fin
```

Una vez más, podemos ver el código completo en este mismo documento en el Anexo II.

4.6.3 Resultados

En los casos anteriores hemos mostrado gráficos con los tiempos de ejecución detallada y, después, el gráfico que unifica el tiempo de ejecución total en el *clúster* variando el número de nodos. En este caso, únicamente hemos considerado de relevancia el gráfico que unifica los tiempos de ejecución, sin mostrar detalles del tiempo de cada etapa. La Figura 27 recoge esta información. Se obtienen las siguientes conclusiones:

- El rendimiento obtenido con la **implementación en C** es paupérrimo: procesar 5 imágenes en C es 72 veces más lento que procesarlas en el *clúster* con un único nodo. Recordemos que el código en C se ejecuta sobre el procesador de la placa, un ARM Cortex A9 de dos núcleos; mientras que el código en *OpenCL* es ejecutado sobre la FPGA. Esto refleja que la potencia de la DE1-SOC viene dada por la FPGA. Una manera de aumentar el rendimiento del programa escrito en C es utilizar las librerías, funciones y macros que se disponen para utilizar los recursos *hardware* de la FPGA desde el código en C. Sin embargo, esto requiere de tiempo y conocimientos; tiempo que se reduce si se implementa el programa en *OpenCL*, ya que el SDK de Altera realiza esta labor de manera automática. El *speedup* que se obtiene respecto a C (representado en la Figura 26) se mantiene bastante constante y aumenta de forma significativa al añadir nodos al *clúster*.

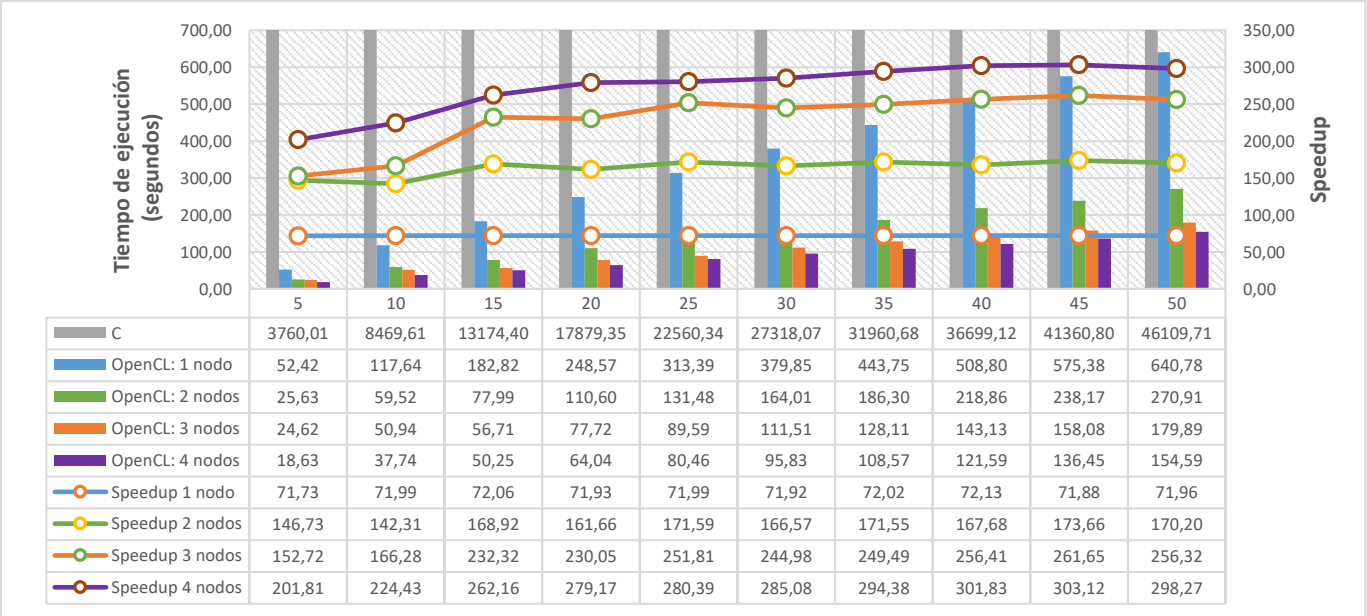
- En el caso de **un nodo**, duplicar el número de imágenes de entrada no duplica el tiempo de ejecución en *OpenCL*; sino que lo duplica y lo supera. Por ejemplo, de procesar 25 a 50 imágenes el tiempo de ejecución se incrementa en un 204%. Esto se debe a los retardos de copiar el programa y el *kernel* en cada ejecución, así como las comprobaciones que se hace del fichero *state.*

- Al introducir un nodo más en el *clúster*, **2 en total**, conseguimos reducir el tiempo de ejecución en *OpenCL* a algo menos de la mitad. En el caso de N=50 el *speedup* es de 2,36, es decir, se ha ejecutado 2 veces y casi la mitad más rápido que con 1 nodo. Lo que está pasando es que, enviar los datos en paralelo y a solamente dos placas, no provoca ningún cuello de botella en la red y, por lo tanto, las ejecuciones de una y otra placa se realizan prácticamente en paralelo (salvo la primera y la última ejecución, que sólo se realizan en una placa a la vez).

- Al utilizar **un tercer nodo** no se obtienen grandes ganancias respecto al caso anterior. Sobre todo, esto se da cuando el número de imágenes a procesar es bajo. Aumentando este número hasta 50 sí observamos una mejora sustancial. El *speedup* es de 3,56, es decir, el programa se ejecuta 3 veces y media más rápido. En este caso parece que la red tampoco actúa de cuello de botella, sin embargo, la mejora del rendimiento se produce cuando el *pipe* se ha llenado. Es decir, cuando las 3 placas tienen sus datos listos y comienzan a ejecutar el programa. Esta es la razón por la que, en el caso de N=5, el tiempo apenas mejora respecto al caso con 2 nodos. Notamos como en los casos de N impar la mejora respecto al caso con 2 nodos es mayor. Por ejemplo, para N=15 el tiempo disminuye unos 21 segundos (77-56), mientras que en el caso N=20 lo hace en 33 segundos (110-77).

- Por último, ejecutando el algoritmo en el *clúster* con **4 nodos** los tiempos de *OpenCL* sí mejoran significativamente en todos los casos, incluso para procesar 5 imágenes. Al contrario que antes, al comparar los tiempos con 3 nodos, se reducen más en los casos de N par. Por ejemplo, para N=35 el tiempo re reduce en 20 segundos (138-108), mientras que para N=40 re reduce en 22 segundos (143-121). Respecto al caso con un nodo y N=50, el *speedup* obtenido es de 4,14, es decir, se ejecuta algo más de 4 veces más rápido. Para N=25 el *speedup* es de 3,89.

- De forma general se puede concluir que añadir nodos al *clúster* disminuye considerablemente el tiempo de ejecución de *OpenCL* a partir de un valor concreto de N, siempre y cuando lo comparemos con un sistema de un único nodo. Por ejemplo, si consideramos a 1.5, es decir una mejora del 50%, un *speedup* aceptable y teniendo en cuenta el coste asociado a añadir un nodo, pasar de 2 a 3 nodos sólo merecería la pena si N es mayor o igual que 50.



	5	10	15	20	25	30	35	40	45	50
C	3760,01	8469,61	13174,40	17879,35	22560,34	27318,07	31960,68	36699,12	41360,80	46109,71
OpenCL: 1 nodo	52,42	117,64	182,82	248,57	313,39	379,85	443,75	508,80	575,38	640,78
OpenCL: 2 nodos	25,63	59,52	77,99	110,60	131,48	164,01	186,30	218,86	238,17	270,91
OpenCL: 3 nodos	24,62	50,94	56,71	77,72	89,59	111,51	128,11	143,13	158,08	179,89
OpenCL: 4 nodos	18,63	37,74	50,25	64,04	80,46	95,83	108,57	121,59	136,45	154,59
Speedup 1 nodo	71,73	71,99	72,06	71,93	71,99	71,92	72,02	72,13	71,88	71,96
Speedup 2 nodos	146,73	142,31	168,92	161,66	171,59	166,57	171,55	167,68	173,66	170,20
Speedup 3 nodos	152,72	166,28	232,32	230,05	251,81	244,98	249,49	256,41	261,65	256,32
Speedup 4 nodos	201,81	224,43	262,16	279,17	280,39	285,08	294,38	301,83	303,12	298,27

Figura 27: Comparativa de la ejecución del algoritmo del flujo óptico con 1 a 4 nodos

4.7 Experimento V: Multifunction printer error diffusion

El error de difusión en impresoras multifunción [28] es un programa que implementa una de las etapas en el proceso de imprimir un documento, como se puede observar en la Figura 28.

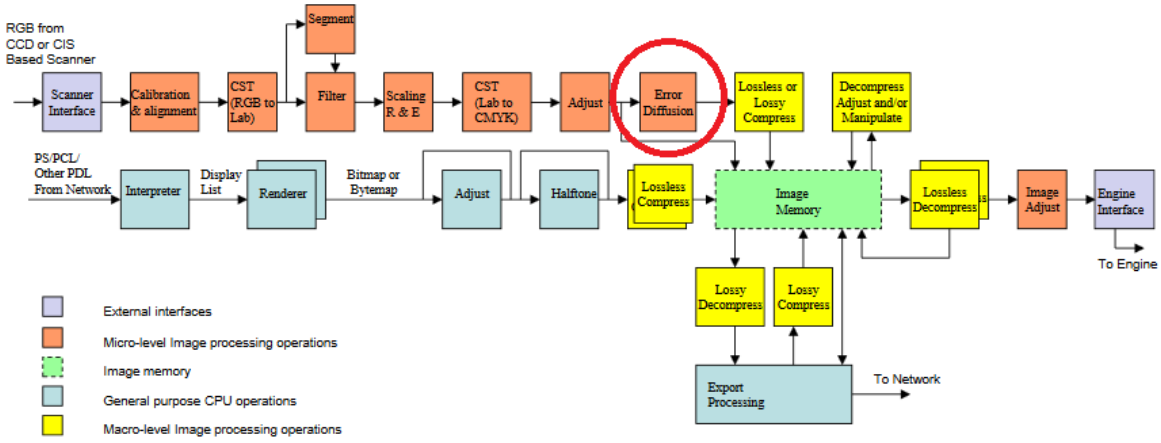


Figura 28: Pipeline en impresoras multifunción

Este programa ha sido creado partiendo del ejemplo de diseño "*Multifunction Printer error diffusion*" que se puede descargar nuevamente desde la web de Altera [29], eliminado los *benchmarks* que se hacen para comparar el rendimiento de la FPGA con su ejecución en C en el procesador ARM de la placa. El programa recibe como entrada una imagen en formato TIFF[10] con una resolución de 600 DPI y colores en formato CMYK, que es el formato típico de colores que manejan la mayoría de impresoras de la actualizad (azul, magenta y amarillo); y genera esa misma imagen con los píxeles semidefinidos, por llamarlos de alguna manera. Es decir, genera la imagen que recibiría una impresora para poder ser imprimida. La Figura 29 ilustra mejor esta idea. La imagen superior es un fragmento ampliado de la imagen original y la imagen inferior es el resultado. La imagen de entrada tiene una resolución de 5100 x 3536 píxeles y ocupa 70 megabytes aproximadamente.

Figura 29: Imagen (ampliada) resultado de aplicar el algoritmo Multifunction printer error diffusion

[10] http://www.fileformat.info/format/tiff/egff.htm

4.7.1 Código del kernel de *OpenCL*

Como comentan en la web de Altera, el código es una variación del algoritmo de error de difusión de Floyd Steinberg, que simplifica las operaciones aritméticas para poder ser ejecutado en *FPGAs* de gama baja. Dada su complejidad, no se va a explicar el código en este documento.

4.7.2 Lanzador del algoritmo en el *clúster*

El lanzador de este programa es muy parecido al lanzador del Flujo óptico, pero con la diferencia de que las imágenes se procesan de una en una. Por lo tanto, la única línea que cambia es la llamada al *script launch.sh* que está en todos los nodos:

Cambiando la siguiente línea del código lanzador del programa del Flujo óptico:

```
ssh root@${NODES[j]} "./launch.sh $OUTPATH $1 -input1=$INPUT$i$EXT
    -input2=$INPUT$k$EXT" >> cluster_de1soc.log &
```

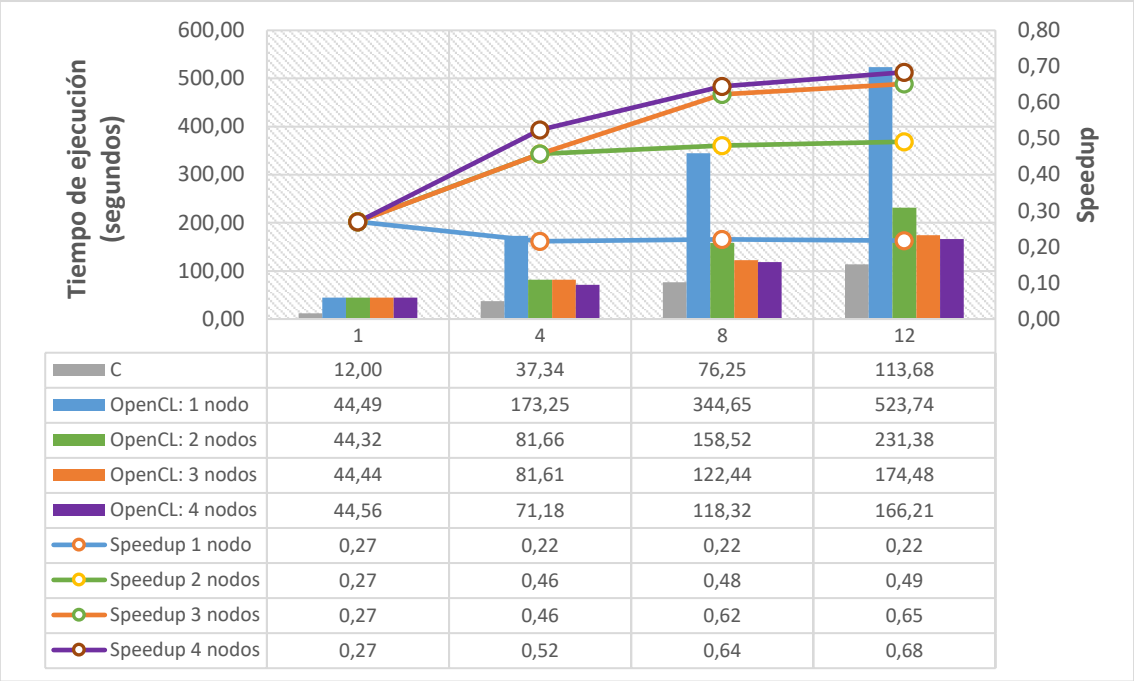
Por esta otra:

```
ssh root@${NODES[j]} "./launch.sh $OUTPATH $1 -i ${INPUT[i]} -o
    out/out_${INPUT[i]} -r -v" >> cluster_de1soc.log &
```

Nos permite reutilizar el mismo lanzador.

4.7.3 Resultados

La Figura 30 contiene las gráficas que representan los tiempos de ejecución de este programa en el *clúster*, variando el número de nodos de 1 a 4. Como las imágenes de entrada son muy pesadas (de 70 megabytes cada una) y, como hemos visto en las anteriores pruebas, la red *Fast Ethernet* es el cuello de botella de nuestro sistema, no hemos considerado importante aumentar excesivamente el número de imágenes de entrada. En este caso las pruebas se han hecho con 1, 4, 8 y 12 imágenes. EL objetivo es procesar ellas en el menor tiempo posible.



	1	4	8	12
C	12,00	37,34	76,25	113,68
OpenCL: 1 nodo	44,49	173,25	344,65	523,74
OpenCL: 2 nodos	44,32	81,66	158,52	231,38
OpenCL: 3 nodos	44,44	81,61	122,44	174,48
OpenCL: 4 nodos	44,56	71,18	118,32	166,21
Speedup 1 nodo	0,27	0,22	0,22	0,22
Speedup 2 nodos	0,27	0,46	0,48	0,49
Speedup 3 nodos	0,27	0,46	0,62	0,65
Speedup 4 nodos	0,27	0,52	0,64	0,68

Figura 30: Comparativa de la ejecución del algoritmo del multifunction printer error diffusion con 1 a 4 nodos

Alcanzamos las siguientes conclusiones:

- EL tiempo de ejecución en C es unas 4 veces más rápido que el código ejecutado en *OpenCL*. Esto se debe a que la ejecución en C se ha hecho directamente sobre la placa, mientras que la ejecución en *OpenCL* se ha hecho en el *clúster* con el programa lanzador y, en este caso, es necesario transmitir la imagen de entrada de 70 MB a los nodos esclavos. En todos los casos, el mayor *speedup* se obtiene con 4 nodos.
- En el caso de procesar una imagen en *OpenCL* no hay mejora entre ejecutarla con un nodo o con varios. Como nos indica la lógica, una imagen se procesa en un nodo y los otros permanecen inactivos.
- Si aumentamos el número de imágenes hasta 4, la verdadera mejora en el tiempo de ejecución en *OpenCL* se consigue cuando pasamos de 1 nodo a 2. En este caso se consigue un *speedup* de 2,12, es decir, el programa se ejecuta algo más del doble de rápido. Añadiendo un tercer nodo no obtenemos absolutamente ninguna mejora en el tiempo, y esto se produce porque uno de los nodos tiene que procesar 2 imágenes y los otros nodos tan solo una. Se procesan 3 imágenes en paralelo (una por nodo) y la última se ejecuta en un solo nodo, los otros 2 esperan a que acabe. La Figura 31 ilustra mejor éste fenómeno.

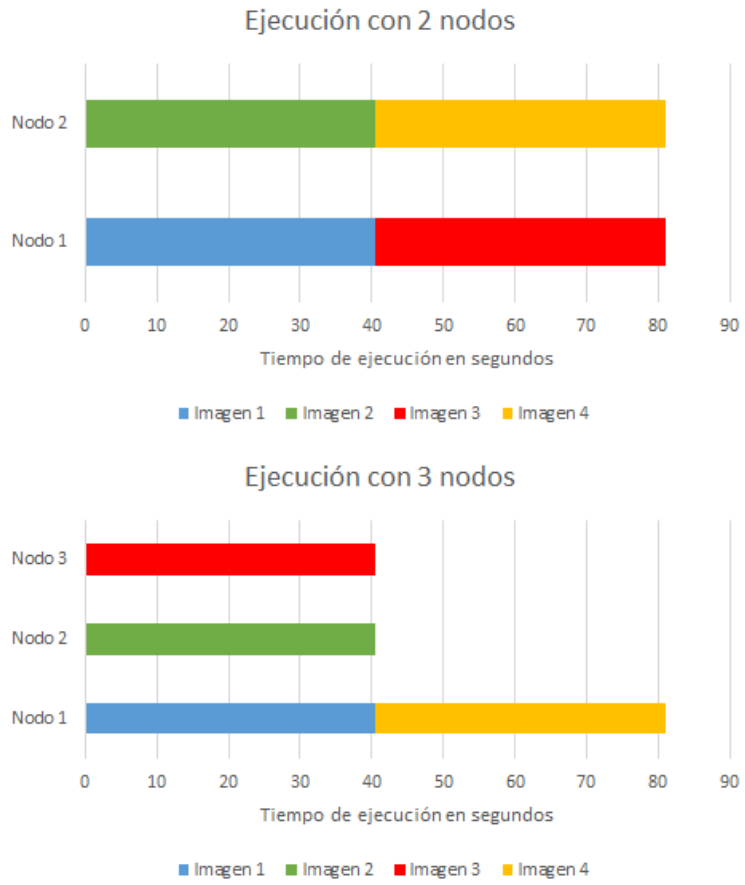


Figura 31: Comparación del tiempo de ejecución del programa Multifunction printer error diffusion con 2 y 3 nodos

- En el resto de los casos el tiempo aumenta de forma lineal. Por ejemplo, en el caso de un nodo, pasar de 4 imágenes a 8 multiplica por dos el tiempo de ejecución de procesar 4 imágenes.

- La conclusión que más nos vuelve a impactar es que la mejora más grande se produce al pasar de 1 nodo a 2. Y que, al añadir un tercer y cuarto nodo, el tiempo apenas se reduce. Como hemos comentado en varias ocasiones, esto se debe a la lentitud de la red de conexión. Pese a que las DE1-SOC tienen una interfaz de red *Gigabit*, la velocidad se adapta a la de *Fast Ethernet*, puesto que tanto los *routers* como el nodo maestro implementan este estándar. Cuando se utiliza un único nodo, los datos se envían a la mayor velocidad que proporciona el ancho de banda, que es de unos 10 MBps. Cuando añadimos un segundo nodo la velocidad se reparte entre ellos, aunque no de manera equitativa. Haciendo pruebas hemos comprobado cómo la velocidad de copia a un nodo es de unos 7 MBps, mientras que la del otro oscila entre los 3 y 4 MBps. Sin embargo, cuando los datos se terminan de pasar al primer nodo, la velocidad del segundo nodo asciende a 10 MBps. Por lo tanto, la latencia de la copia del segundo nodo se solapa con el tiempo de ejecución del primero. Por el contrario, esto no ocurre cuando añadimos más nodos. Con 3 y 4 nodos la copia de los datos en paralelo hace que la velocidad de copia de cada nodo sea muy desproporcionada. El nodo 1 alcanza una velocidad de 6 MBps, el nodo 2 de 3 MBps y los nodos 3 y 4 oscilan entre 1 y 2 MBps. Y, al igual que antes, cuando se termina de enviar los datos a un nodo, la velocidad del resto aumenta. Por desgracia, la ejecución en paralelo no compensa esta pérdida de velocidad en la red; ya que es muy probable que la red siempre esté saturada con, al menos, 3 nodos y que

haga de cuello de botella. La Figura 32 ilustra de manera gráfica este problema (el tiempo de copia de los datos se representa en gris y se ha obviado, por simplicidad, la copia de los resultados al nodo maestro).

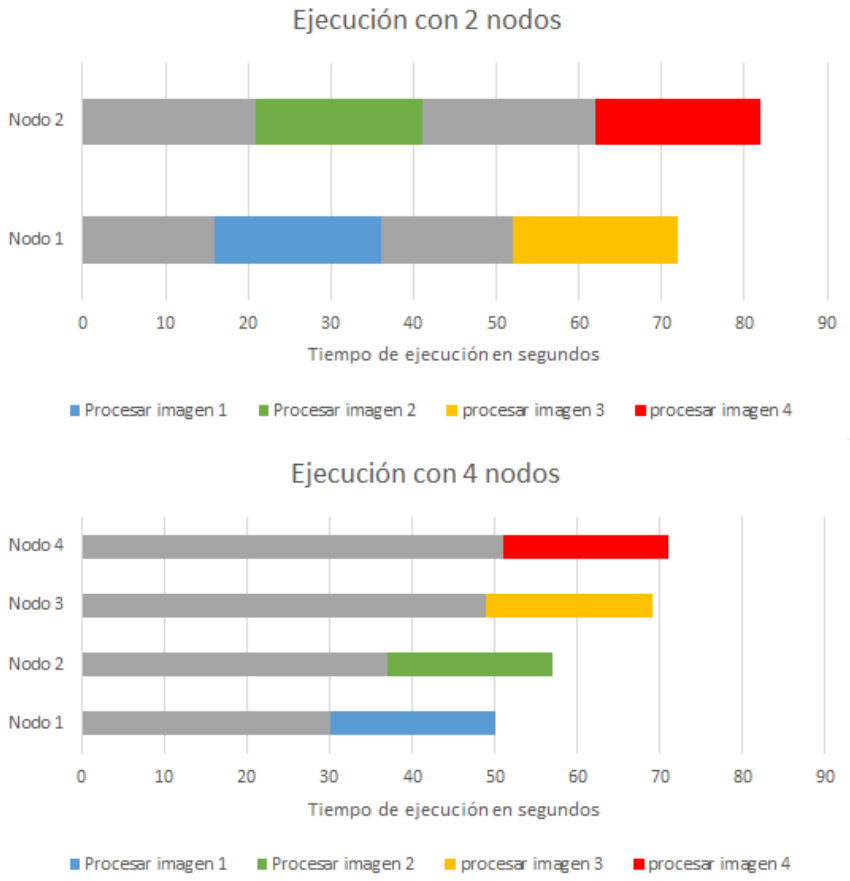


Figura 32: Cuello de botella en la red al ejecutar el programa del Multifuncion printer error diffusion en el clúster con 2 y 4 nodos

4.8 Comparativa con una *Workstation*

En este apartado vamos a comparar y analizar los tiempos que se han obtenido al ejecutar los programas que forman la suite en un equipo de altas prestaciones, una *Workstation*, con los obtenidos en el *clúster* de DE1-SOCs. Esta *Workstation* es uno de los equipos que cuenta el Departamento de Arquitectura de Computadores y Automática (DACYA) para la realización de sus experimentos. Cuenta con dos procesadores Xeon E5-2695 v3, cada uno con 14 *cores* físicos y 2 *threads/core*, lo que hace un total de 56 *cores*. También cuenta con 16 GB de memoria RAM y una unidad de almacenamiento de estado sólido o SSD. Podría parecernos que la comparación no será justa, pero tenemos que pensar que las placas son muchísimo más eficientes energéticamente y que el coste del clúster es menor al coste de la *Workstation*. Cada una de las placas tiene un coste aproximado de 250 dólares americanos ($175 para universidades), mientras que el coste de uno sólo de los Xeon E5-2695 v3 de la *Workstation* tiene un precio

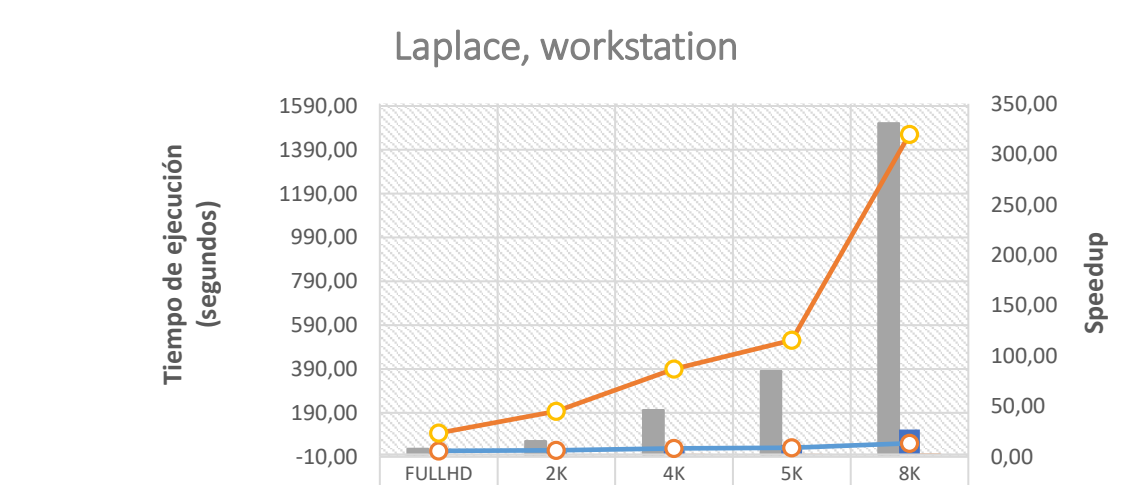
recomendado de 2400 dólares americanos. No obstante, veremos un análisis del consumo y costes del *clúster* y de la *Workstation* más adelante en este apartado.

Para las pruebas, se ha utilizado el compilador de Intel de *OpenCL* en su versión 1.2.3. Se han modificado los programas de la suite para poder ser compilados y ejecutados en la *Workstation*. En algunos casos, los programas se han optimizado para poder utilizar todo el *hardware* de la *Workstation* y en otros se ha ejecutado exactamente el mismo código que en las placas.

4.8.1 Experimento I: Detección de bordes con el operador de Laplace

La ejecución de este programa se ha hecho tanto sin alterar la versión del *clúster* (color azul), como con una versión optimizada (color naranja). Aunque no se nombró, el algoritmo de *Laplace* estaba creado para aprovechar el potencial de la FPGA de la DE1-SOC. Se utilizaba un único grupo de trabajo de *OpenCL* [30]. Esta es la versión sin alterar. Sin embargo, para aprovechar el potencial de la *Workstation*, se ha modificado el código del *kernel* de *OpenCL* para utilizar varios grupos de trabajo. El número de ellos ha sido dado por el propio compilador en tiempo de compilación. Esta es la versión optimizada.

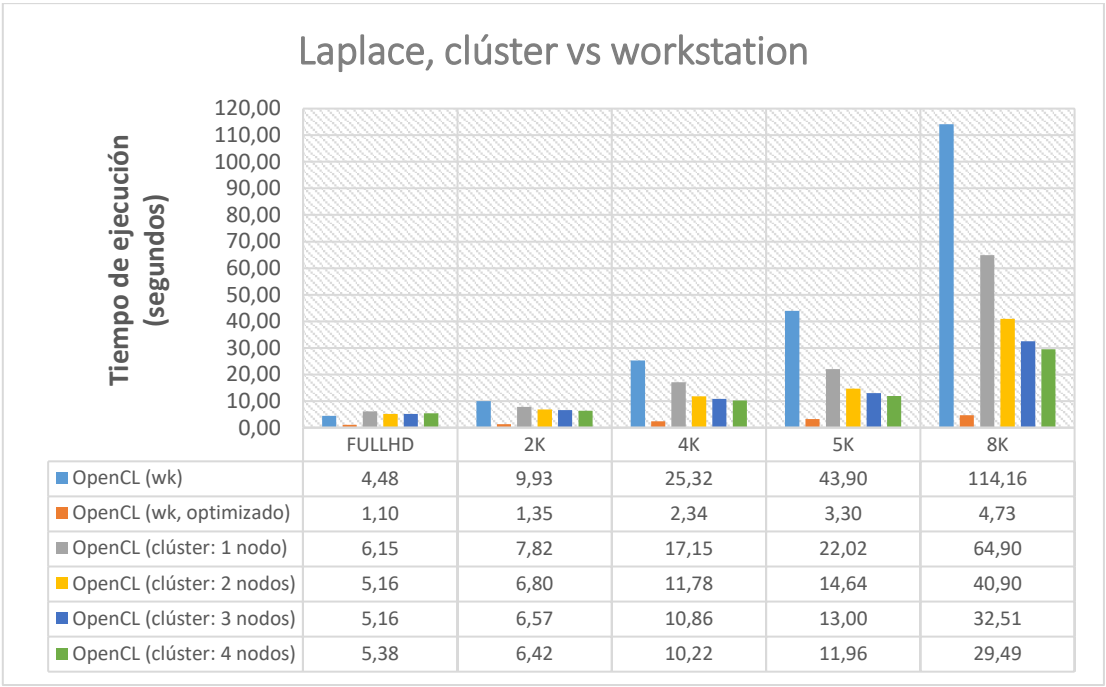
Como se observa en la Figura 33, la ejecución del algoritmo en C es varias veces más lenta que en cualquiera de las versiones en *OpenCL*, cosa que no tendría por qué ser así; como veremos en otros casos. La versión de *OpenCL* sin optimizar sufre un incremento considerable del tiempo de ejecución al aumentar el tamaño de la imagen de entrada. Sin embargo, en la versión optimizada de *OpenCL* este incremento es más moderado. El *speedup* aumenta al aumentar la resolución de la imagen de entrada. En el caso de *OpenCL* sin optimizar es casi constante, mientras que en el caso optimizado la curva crece con gran rapidez.



	FULLHD	2K	4K	5K	8K
C	25,60	60,61	203,25	381,25	1511,22
OpenCL	4,48	9,93	25,32	43,90	114,16
OpenCL (optimizado)	1,10	1,35	2,34	3,30	4,73
Speedup OpenCL	5,72	6,10	8,03	8,69	13,24
Speedup OpenCL (optimizado)	23,36	44,90	86,82	115,58	319,71

Figura 33: Workstation: tiempos de ejecución Laplace. OpenCl vs C

La Figura 34 recoge los tiempos de ejecutar el algoritmo de *Laplace* en *OpenCL* en la *Workstation* y en el *clúster* de placas variando el número de nodos. Nótese que la abreviatura "wk" que aparece en las tablas hace referencia a la *Workstation.* Se puede observar que los tiempos de ejecución entre el *clúster* y la *Workstation* son muy parecidos en la versión con un único grupo de trabajo para tamaños de imagen pequeños (inferiores a *FullHD*). Para tamaños mayores a *FullHD* el tiempo de ejecución del *clúster* con un único nodo ya supera al de la *Workstation*. Esto se debe a que, en este caso particular, la FPGA supera en rendimiento a uno de los *cores* de la *Workstation.* Sin embargo, esto cambia al aumentar el número de grupos de trabajo. En este caso, ni siquiera con 4 nodos el *clúster* es capaz de igualar el tiempo de ejecución de la *Workstation* (1,09 vs 5,37 segundos, respectivamente). Es posible que el retardo y la saturación de la red haya tenido mucho que ver, pero a falta de contar con una red de mejores prestaciones con la que comparar, no se puede afirmar con seguridad. Puede que se deba a que el *hardware* de la FPGA no supere al de la *Workstation* cuando se emplea todo su potencial. Este percance nos incapacita para determinar el número de nodos que serían necesarios para igualar los tiempos de la *Workstation*.

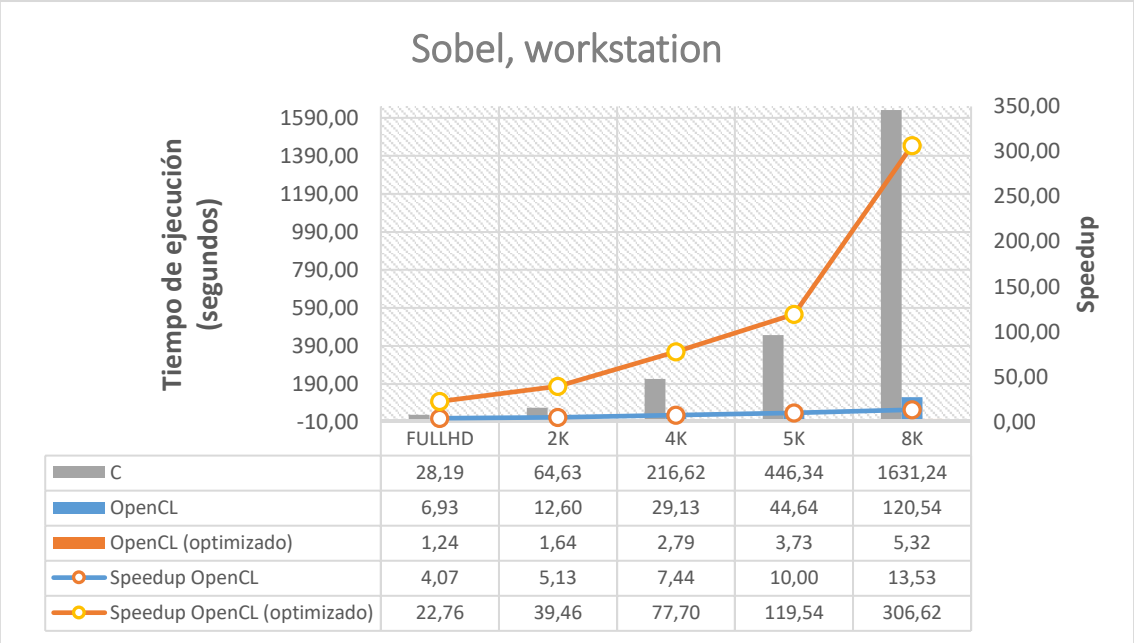


	FULLHD	2K	4K	5K	8K
OpenCL (wk)	4,48	9,93	25,32	43,90	114,16
OpenCL (wk, optimizado)	1,10	1,35	2,34	3,30	4,73
OpenCL (clúster: 1 nodo)	6,15	7,82	17,15	22,02	64,90
OpenCL (clúster: 2 nodos)	5,16	6,80	11,78	14,64	40,90
OpenCL (clúster: 3 nodos)	5,16	6,57	10,86	13,00	32,51
OpenCL (clúster: 4 nodos)	5,38	6,42	10,22	11,96	29,49

Figura 34: Laplace: Workstation vs Clúster

4.8.2 Experimento II: Detección de bordes con el operador de Sobel

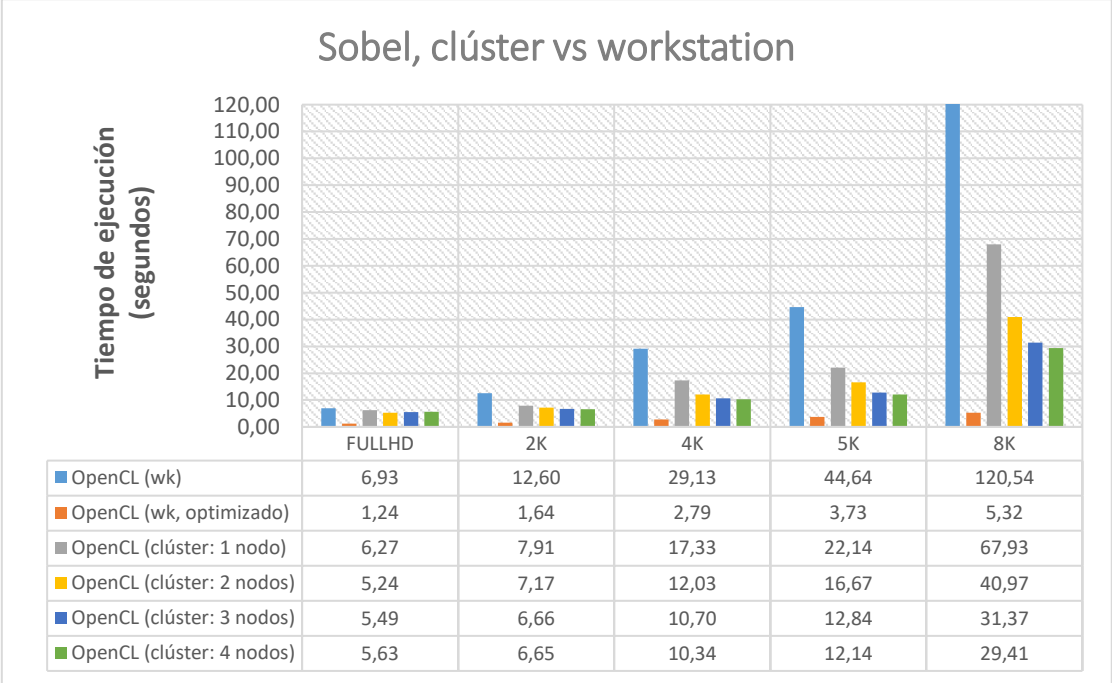
Este caso se parece al de *Laplace*, dado que ambos algoritmos comparten código y tienen una complejidad casi idéntica. En la Figura 34 se observa que la ejecución en *OpenCL* es bastante más rápida que la ejecución en C. La ganancia obtenida o *speedup* entre la versión de *OpenCL* original frente a la optimizada vuelve a ser mayor al aumentar el tamaño de la imagen de entrada. El *speedup* de *OpenCL* respecto a C (representado en la Figura 35) es semejante al caso de *Laplace*: es casi constante en la versión sin optimizar y de rápido crecimiento en la versión optimizada.



	FULLHD	2K	4K	5K	8K
C	28,19	64,63	216,62	446,34	1631,24
OpenCL	6,93	12,60	29,13	44,64	120,54
OpenCL (optimizado)	1,24	1,64	2,79	3,73	5,32
Speedup OpenCL	4,07	5,13	7,44	10,00	13,53
Speedup OpenCL (optimizado)	22,76	39,46	77,70	119,54	306,62

Figura 35: Workstation: tiempos de ejecución Sobel. OpenCl vs C

Como en el caso de *Laplace*, la ejecución en la *Workstation* en *OpenCL* con un único grupo de trabajo es más lenta que el clúster con un solo nodo. Y, si tomamos como referencia la versión que emplea múltiples grupos de trabajo, el tiempo del *clúster* es muy superior incluso empleando los 4 nodos de los que hemos dispuesto. Una vez más es difícil inferir el número de nodos que harían falta para igualar los tiempos dados por la *Workstation*, ya que la mejora al utilizar 3 o 4 nodos es prácticamente nula. Sería necesario utilizar una red *Gigabit* para determinar este valor. La Figura 36 recoge los tiempos que acabamos de comentar.



	FULLHD	2K	4K	5K	8K
OpenCL (wk)	6,93	12,60	29,13	44,64	120,54
OpenCL (wk, optimizado)	1,24	1,64	2,79	3,73	5,32
OpenCL (clúster: 1 nodo)	6,27	7,91	17,33	22,14	67,93
OpenCL (clúster: 2 nodos)	5,24	7,17	12,03	16,67	40,97
OpenCL (clúster: 3 nodos)	5,49	6,66	10,70	12,84	31,37
OpenCL (clúster: 4 nodos)	5,63	6,65	10,34	12,14	29,41

Figura 36: Sobel: Workstation vs Clúster

4.8.3 Experimento III: Emborronamiento Gaussiano

Como vemos en la Figura 37, los tiempos de ejecución del programa del Emborronamiento Gaussiano son muy semejantes tanto en C como en *OpenCL* en la *Workstation*. En todo caso se obtiene un *speedup* mayor a 1,3, es decir, el programa se ejecuta un 30% más rápido. La razón de ello es que el código no se ha optimizado para ser ejecutado en la FPGA, simplemente se ha implementado para ser ejecutado en ella. Tampoco se han empleado más grupos de trabajo en la versión de *OpenCL*. No obstante, con imágenes de alta resolución sí observamos una diferencia notable. Obtenemos un *speedup* cercano a 2, es decir, una reducción del tiempo de ejecución a la mitad. Un caso curioso es el que supone pasar de la imagen 4k (3840x2160 píxeles) a la 5k (5120x2160 píxeles): en C el tiempo aumenta 13 segundos mientras que en *OpenCL* aumenta en 16 segundos. Es curioso porque solo ocurre con este par de imágenes y porque la ejecución en C es siempre más lenta.

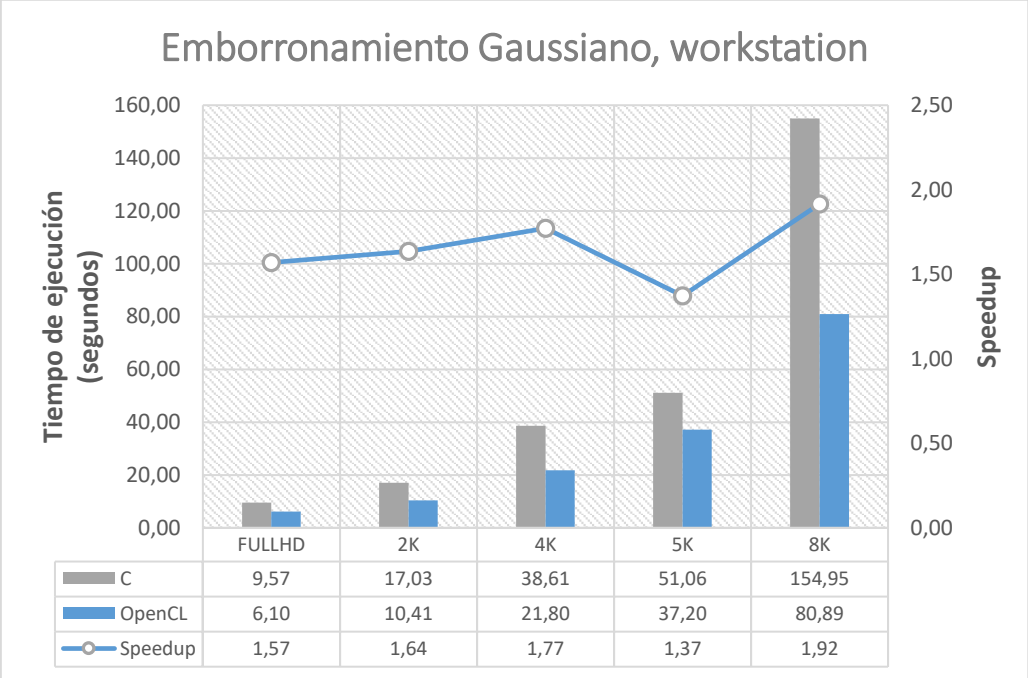
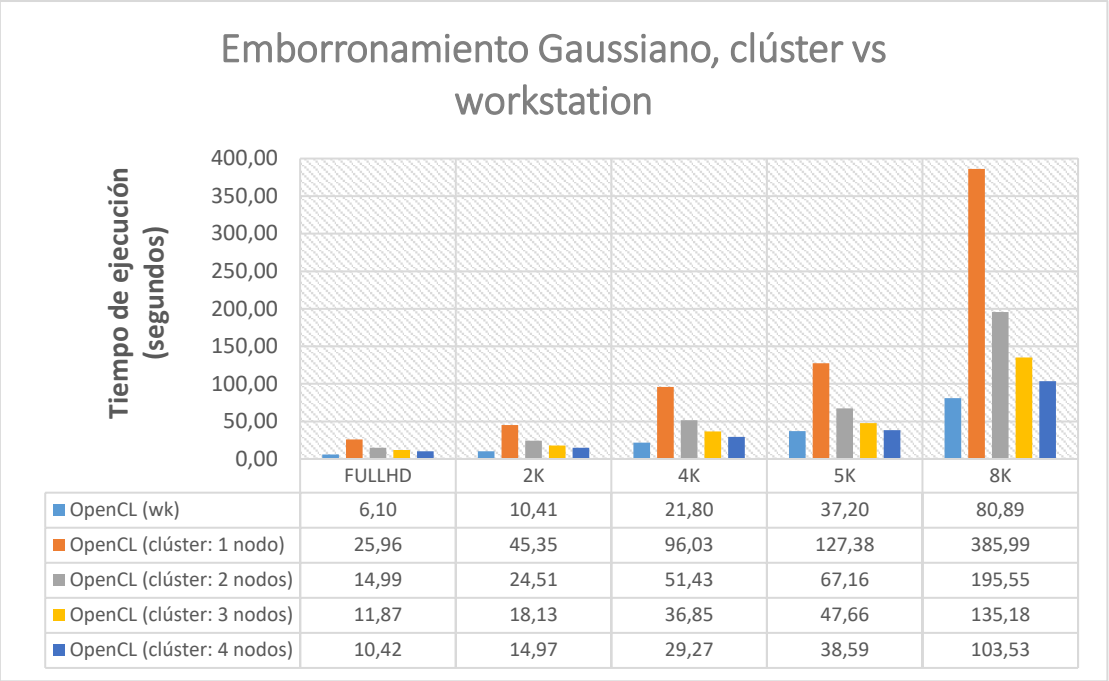


Figura 37: Workstation: tiempos de ejecución Emborronamiento Gaussiano. OpenCl vs C

Como se percibe en la Figura 38, en cualquiera de los casos la *Workstation* ejecuta el programa en un tiempo menor al del clúster. Los tiempos más parecidos se han obtenido al procesar la imagen 5k (5120x2160 píxeles) en el *clúster* con 4 nodos, en el que apenas hay una diferencia de 1 segundo. A diferencia de en casos anteriores, pasar de 3 a 4 nodos en el *clúster* sí supone una mejora significativa en el tiempo de ejecución: salvo en la imagen de resolución *FullHD,* el *speedup* es cercano a 1,25.



	FULLHD	2K	4K	5K	8K
OpenCL (wk)	6,10	10,41	21,80	37,20	80,89
OpenCL (clúster: 1 nodo)	25,96	45,35	96,03	127,38	385,99
OpenCL (clúster: 2 nodos)	14,99	24,51	51,43	67,16	195,55
OpenCL (clúster: 3 nodos)	11,87	18,13	36,85	47,66	135,18
OpenCL (clúster: 4 nodos)	10,42	14,97	29,27	38,59	103,53

Figura 38: Emborronamiento Gaussiano: Workstation vs Clúster

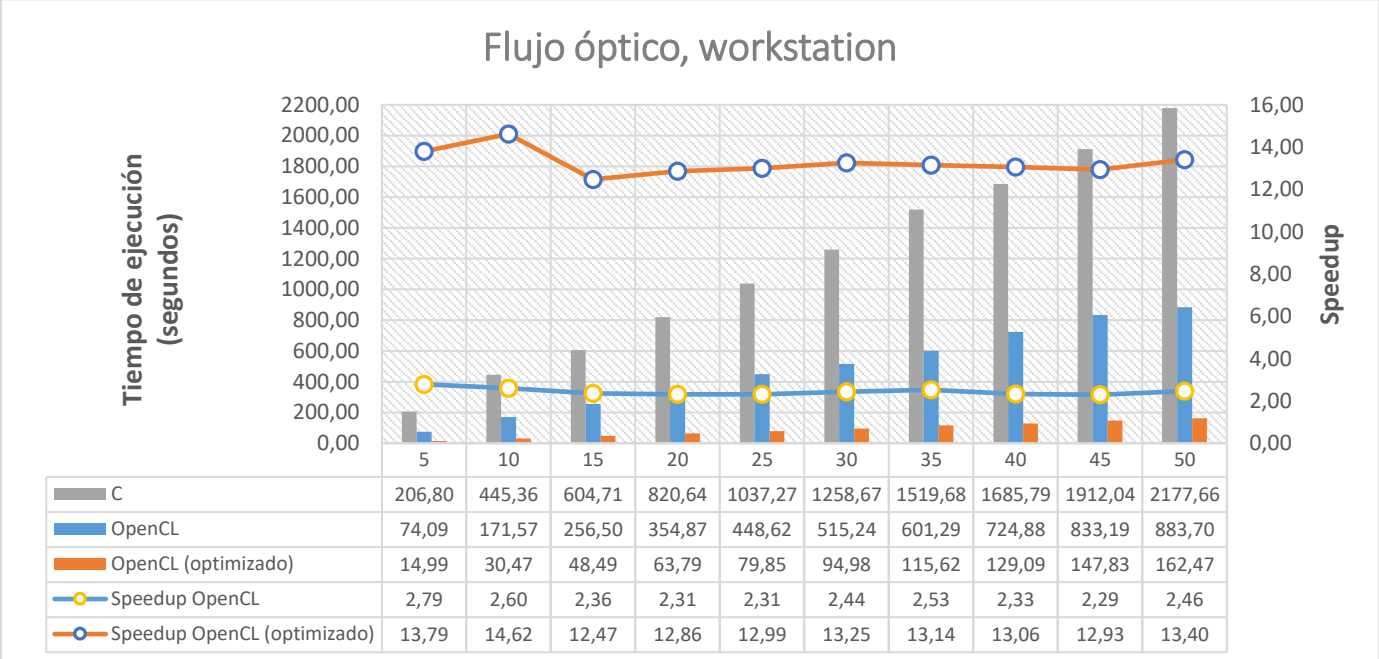
Teniendo en cuenta el caso de mayor entrada, el de la imagen 8k, y de manera aproximada; se necesitarían 6 ó 7 nodos para igualar el tiempo de ejecución de la *Workstation.* La Tabla 3 contiene los cálculos que hemos realizados para llegar a esta conclusión: al añadir un nodo, la diferencia del tiempo de ejecución entre éste y el anterior se reduce aproximadamente a la mitad. Si la tendencia se mantiene, para que esta diferencia valga 80 (el tiempo en la *Workstation*), tenemos que repetir esta operación 2 veces.

	Tiempo ejecución. Imagen 8k	$T_{i\ nodos} - T_{i+1\ nodos}$
OpenCL (wk)	80,894	-
OpenCL (clúster: 1 nodo)	$T_1 = 385{,}99$	$T_1 - T_2 = 385{,}99 - 195{,}54 = 190{,}44$
OpenCL (clúster: 2 nodos)	$T_2 = 195{,}54$	$T_2 - T_3 = 195{,}54 - 135{,}18 = 60{,}36$
OpenCL (clúster: 3 nodos)	$T_3 = 135{,}18$	$T_3 - T_4 = 135{,}18 - 103{,}52 = 31{,}65$
OpenCL (clúster: 4 nodos)	$T_4 = 103{,}52$	$T_4 - T_5 = 103{,}52 - T_5 \approx 15 \Rightarrow T_5 = 88$
OpenCL (clúster: 5 nodos)	$T_5 \approx 88$	$T_5 - T_6 = 88 - T_6 \approx 8 \Rightarrow T_6 = 80$
OpenCL (clúster: 6 nodos)	$T_6 \approx 80$	-

Tabla 3: Emborronamiento Gaussiano. Cálculo aproximado del número de nodos necesarios para igualar a la Workstation

4.8.4 Experimento IV: El algoritmo del Flujo óptico

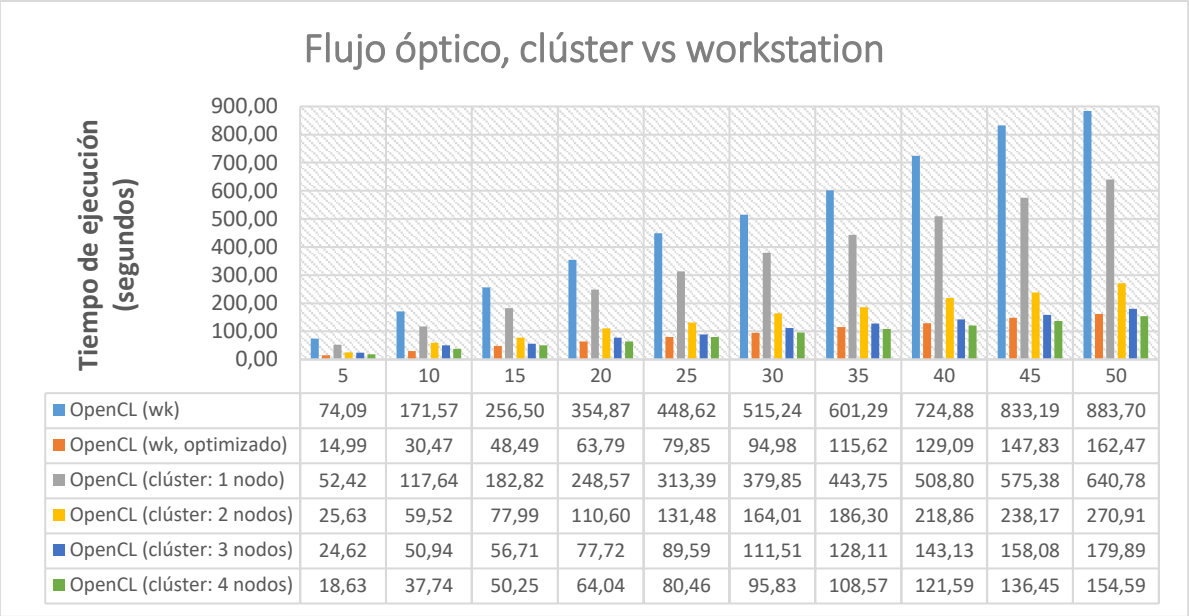
En este caso, como se puede distinguir en la Figura 39, los tiempos de ejecución en C superan con un gran margen a los tiempos de *OpenCL*. En este caso se ha vuelto a optimizar el código de *OpenCL* ejecutado en la *Workstation* para obtener unos tiempos que puedan rivalizar con el *clúster* de FPGAs. Nuevamente se ha modificado el código del *kernel* de *OpenCL* para utilizar más grupos de trabajo y aprovechar así el *hardware de la Workstation.* En este caso observamos un crecimiento lineal: el tiempo de ejecución aumenta en consonancia al aumento del número de imágenes a procesar. En la mayoría de los casos el *speedup* obtenido es cercano o superior a 2,40, lo cual significa que la versión original de *OpenCL* procesa las imágenes en algo menos de la mitad del tiempo que tarda la versión en C. En la versión optimizada este *speedup* es cercano a 13, valor que indica con claridad la ventaja de dedicar tiempo a optimizar el código de *OpenCL*.



	5	10	15	20	25	30	35	40	45	50
C	206,80	445,36	604,71	820,64	1037,27	1258,67	1519,68	1685,79	1912,04	2177,66
OpenCL	74,09	171,57	256,50	354,87	448,62	515,24	601,29	724,88	833,19	883,70
OpenCL (optimizado)	14,99	30,47	48,49	63,79	79,85	94,98	115,62	129,09	147,83	162,47
Speedup OpenCL	2,79	2,60	2,36	2,31	2,31	2,44	2,53	2,33	2,29	2,46
Speedup OpenCL (optimizado)	13,79	14,62	12,47	12,86	12,99	13,25	13,14	13,06	12,93	13,40

Figura 39: Workstation: tiempos de ejecución Flujo óptico. OpenCl vs C

Si lo comparamos con el *clúster* de FPGAs, como se aprecia en la Figura 40, los tiempos de ejecución de la *Workstation* en su versión de *OpenCL* sin optimizar son siempre mayores a los tiempos de la *Workstation* con el código optimizado. También son mayores a los tiempos de ejecución en el *clúster* incluso en el caso de utilizar un solo nodo. Por esta razón vamos a centrar nuestro análisis en el caso optimizado. Recordemos que el objetivo de este programa es procesar un conjunto formado por N imágenes en el menor tiempo posible, imágenes que se procesan en grupos de 2. Observamos cómo la *Workstation* realiza esta tarea más rápido que el *clúster* si se procesan menos de 35 imágenes. Y como, a partir de ese valor, es el *clúster* con 4 nodos el que supera a la *Workstation.* Si nos fijamos en los casos del *clúster* con 3 o menor número de nodos, en ninguno de ellos se supera a la *Workstation.*



	5	10	15	20	25	30	35	40	45	50
OpenCL (wk)	74,09	171,57	256,50	354,87	448,62	515,24	601,29	724,88	833,19	883,70
OpenCL (wk, optimizado)	14,99	30,47	48,49	63,79	79,85	94,98	115,62	129,09	147,83	162,47
OpenCL (clúster: 1 nodo)	52,42	117,64	182,82	248,57	313,39	379,85	443,75	508,80	575,38	640,78
OpenCL (clúster: 2 nodos)	25,63	59,52	77,99	110,60	131,48	164,01	186,30	218,86	238,17	270,91
OpenCL (clúster: 3 nodos)	24,62	50,94	56,71	77,72	89,59	111,51	128,11	143,13	158,08	179,89
OpenCL (clúster: 4 nodos)	18,63	37,74	50,25	64,04	80,46	95,83	108,57	121,59	136,45	154,59

Figura 40: Flujo óptico: Workstation vs Clúster

En el caso de la *Workstation* y *OpenCL* optimizado se procesan 2 imágenes cada 3,74 segundos, mientras que el *clúster* con 4 nodos lo hace en 4,65 segundos; 0,91 segundos más por imagen. Añadir más nodos al *clúster* reduciría aún más el tiempo de ejecución y se acercaría al rendimiento obtenido con la *Workstation*, pero con el consecuente aumento del consumo energético y coste del mismo.

De forma aproximada y simplificada, pues no tenemos en cuenta posibles retardos causados por un estado de saturación de la red y según los tiempos obtenidos en el caso de N=50; se puede utilizar una función potencial mediante el método de mínimos cuadrados (color rojo en la Figura 41) que se aproxime a la ecuación que representa los valores medidos (color azul en la Figura 41). Para procesar 2 imágenes por segundo el tiempo total de ejecución de las 50 imágenes tiene que ser 50 segundos, ya que se procesan de 2 en 2. Por lo tanto, serían necesarios entre 11 y 12 nodos para obtener dicho tiempo.

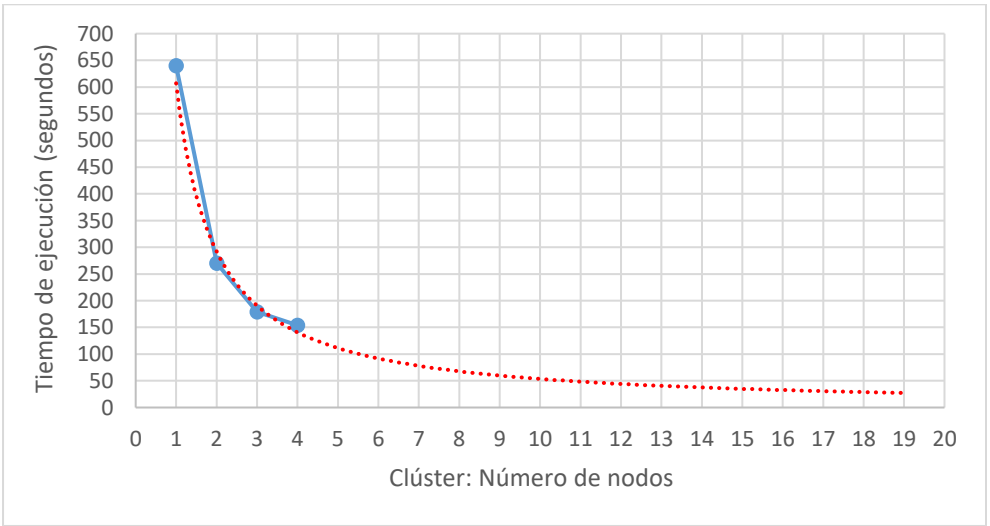


Figura 41: Flujo óptico: Determinación del valor del número de nodos para procesar 2 imágenes por segundo

4.8.5 Experimento V: Multifunction printer error diffusion

Este es el caso que hemos comentado con anterioridad, en el que el tiempo de ejecución en C no es mayor al de *OpenCL*, más bien es todo lo contrario. Como se aprecia en la Figura 42, el tiempo de ejecución en la *Workstation* en *OpenCL* es muy superior al tiempo de ejecución en C en todos los casos de entrada. Hablamos de un tiempo 93, 66, 56 y 50 veces inferior al caso de *OpenCL* al procesar 1, 4, 8 y 12 imágenes respectivamente. Esto queda reflejado en el *speedup,* que es menor a 1 y muy cercano a 0, lo que significa que el tiempo empeora con gravedad.

La pregunta que nos formulamos es por qué ocurre esto. En primer lugar, ocurre que cuando un código se ejecuta en *OpenCL*, es necesario copiar los datos de entrada y salida al espacio de memoria del dispositivo de *OpenCL*; ya sea una CPU o una tarjeta gráfica. Este espacio no es el mismo que el de la memoria principal que maneja el código en C, pese a que el dispositivo elegido sea la CPU, como es en nuestro caso. Este proceso de copia conlleva tiempo. Si el tiempo de procesamiento en C es menor al tiempo de copia más el tiempo de ejecución en el dispositivo en *OpenCL*, es más rápido ejecutar el programa en C directamente. En segundo lugar, esto se debe a un claro desaprovechamiento del *hardware* de la *Workstation.* En este caso se ha implementado y ejecutado el mismo código en C y en *OpenCL,* sin hacer uso de los grupos de trabajo. No se han podido implementar dada la naturaleza del código del *kernel*, que hace imposible su uso a menos que se cambiase el código por completo. Esto último haría que la comparativa con el *clúster* fuera menos justa.

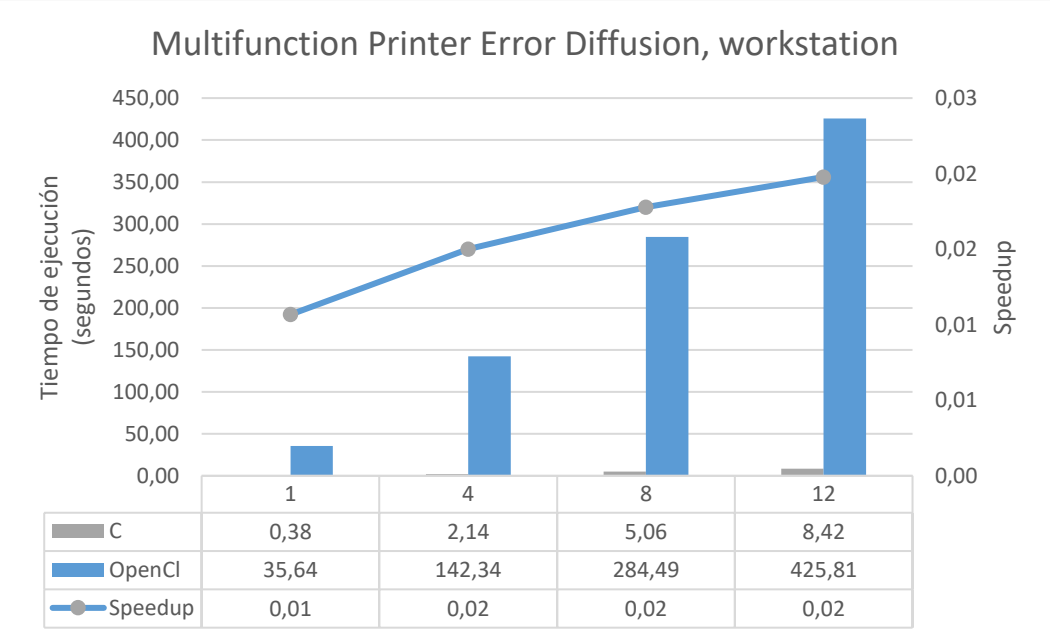
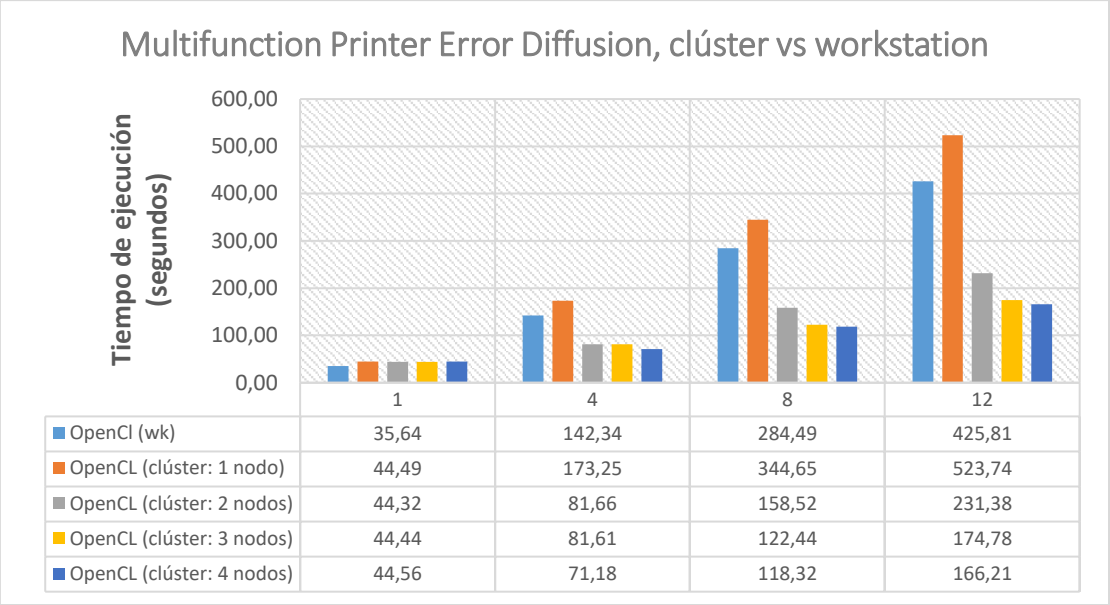


Figura 42: Workstation: tiempos de ejecución Error printer. OpenCl vs C

La Figura 43 presenta la comparativa entre la *Workstation* y el clúster, variando el número de nodos, con los tiempos de ejecución en *OpenCL*. Salvo en el caso de procesar una sola imagen, en el que el *clúster* solamente aprovecha uno de los nodos por cómo se ha implementado la paralización en el programa lanzador, el *clúster* con 2 nodos procesa las imágenes de entrada en un tiempo inferior a la *Workstation.* En todos los casos se procesan casi en la mitad de tiempo, o lo que es lo mismo, el *speedup* es cercano a 2. Por lo tanto, en este caso 2 nodos serían suficientes para superar a la *Workstation.* Si se pudieran utilizar más grupos de

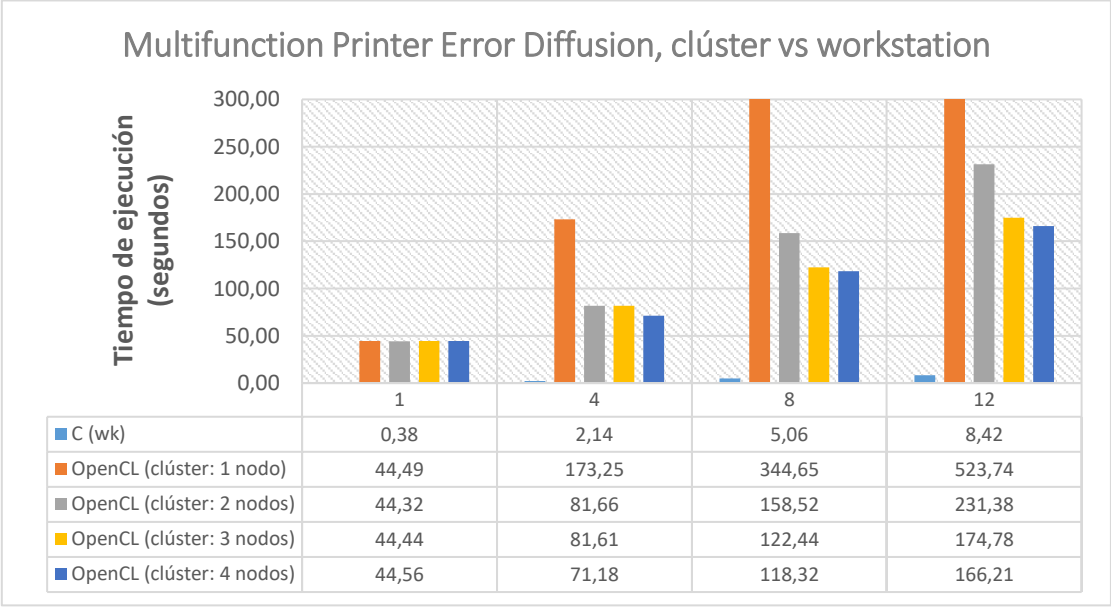
trabajo en la *Workstation*, el tiempo de ejecución se reduciría en gran medida y serían necesarios más nodos en el *clúster* para igualarlo. Pero esto es algo que no se puede comprobar sin modificar los códigos del *kernel*.



	1	4	8	12
OpenCl (wk)	35,64	142,34	284,49	425,81
OpenCL (clúster: 1 nodo)	44,49	173,25	344,65	523,74
OpenCL (clúster: 2 nodos)	44,32	81,66	158,52	231,38
OpenCL (clúster: 3 nodos)	44,44	81,61	122,44	174,78
OpenCL (clúster: 4 nodos)	44,56	71,18	118,32	166,21

Figura 43: Error printer: Workstation (OpenCL) vs Clúster (OpenCL)

Sin embargo, si comparamos la ejecución en la *Workstation* en C; ésta resulta ser mucho más rápida que el *clúster* con *OpenCL*. El *clúster* sufre la penalización causada por la alta latencia de la red mientras que, en la *Workstation*, el programa se ha ejecutado de forma local y sólo se ha tenido en cuenta el tiempo de ejecución del programa. Aunque no se ha representado en las gráficas de la Figura 44, el tiempo de ejecución de una imagen en un nodo del *clúster*, ejecutado de manera local sin tener en cuenta la red, es de unos 0,006 segundos; muy inferior a los 0,38 del tiempo de la *Workstation*.



	1	4	8	12
C (wk)	0,38	2,14	5,06	8,42
OpenCL (clúster: 1 nodo)	44,49	173,25	344,65	523,74
OpenCL (clúster: 2 nodos)	44,32	81,66	158,52	231,38
OpenCL (clúster: 3 nodos)	44,44	81,61	122,44	174,78
OpenCL (clúster: 4 nodos)	44,56	71,18	118,32	166,21

Figura 44: Error printer: Workstation (C) vs Clúster (OpenCL)

4.8.6 Consumo y coste

El último paso de esta comparativa consiste en calcular los costes de la *Workstation* y del clúster, tanto en consumo energético como en el propio coste de los componentes. Como veremos a continuación, la estimación del coste de los componentes de las *Workstation* se ha calculado como cota inferior, mientras que la del *clúster* se hecho como cota superior. Esto es así por la imposibilidad de medir el consumo de la *Workstation* con instrumentación, dada la ubicación física de la computadora, que es de acceso restringido.

Comencemos por el ***clúster*** de DE1-SOCs. Como vimos en la sección 3.2 Creación del clúster, los elementos que forman nuestro *clúster* son: 4 placas DE1-SOC, dos *routers* y un ordenador portátil. Determinar el consumo de las placas DE1-SOC sin instrumentación resulta algo difícil dada la falta de documentación oficial. Según información del foro oficial de Altera[11], el consumo de la **DE1-SOC** ejecutando una imagen de UNIX/Linux en modo HPS, la conexión Ethernet y los dos *cores* con una carga del 100% de trabajo es de 0.53A. Multiplicado por los 12V de potencial eléctrico nos da 6.36W de energía consumida. Según las especificaciones oficiales dadas por el fabricante, el **ordenador portátil** que actúa de nodo *master* requiere de una fuente de 19v y 6.32A. En mediciones realizadas ejecutando el programa lanzador, se ha medido un consumo medio de energía de 18W. El ***router*** Amper Xavi 7868r tiene una corriente media y voltaje de 1.25A a 12V, un total de 15W de potencia; y el ***router*** TD5130 uno de 1A a 12V, 12W en total. Estos cálculos quedan resumidos en la Tabla 4, habiendo quedado parametrizados en función del número de nodos trabajadores y *routers*.

	Intensidad (A)	Voltaje (V)	Potencia (W)	Potencia clúster N nodos (W)
DE1-SOC	0,53	12	6,36	$\mathbf{6,36N + 15Rx + 12Ro + 18}$
Router Amper Xavi 7868r (Rx)	1,25	12	15	
Router ONO *TD5130 (Ro)*	1	12	12	
Portátil	-	-	18	

Tabla 4: Consumo del clúster con N nodos

Respecto a la ***Workstation***, como hemos comentado, se ha realizado una aproximación mediante una cota inferior. Nos ha sido imposible medir el consumo con una herramienta *software* dado que la máquina carece de los sensores necesarios para tal fin. Tampoco hemos podido realizar mediciones físicas debido a que la *Workstation* está ubicada en un recinto de la facultad de acceso restringido a estudiantes. No obstante, hemos realizado un cálculo aproximado en función de alguno de los componentes de la *Workstation* y el consumo indicado en una web[12] (o la media entre el mínimo y el máximo anunciado en dicha web). No hemos contado el consumo de ventiladores ni tarjetas gráficas. Estos datos quedan se reflejan en la tabla 5.

[11] https://alteraforum.com/forum/showthread.php?t=48618

[12] http://www.buildcomputers.net/power-consumption-of-pc-components.html

	Potencia (W)	Potencia total (W)	TOTAL (W)
CPU[13]	120	$120 * 2 = 240$	**322,368**
RAM [31]	4,35	$4{,}35 * 2 = 8{,}7$	
HDD	6,5 a 9 => 7,75	$7{,}75 * 1 = 7{,}75$	
SDD	0,6 a 2,8 => 1,7	$1{,}7 * 2 = 3{,}4$	
Placa base	45 a 80 => 62,5	$62{,}5 * 1 = 62{,}5$	

Tabla 5: Consumo de la Workstation

Utilizando una web especializada en calcular el consumo en función de los componentes y las horas de funcionamiento de un PC o servidor[14] se ha obtenido un consumo de 325W, como se aprecia en la Figura 45. Este valor es muy cercano al que hemos calculado nosotros.

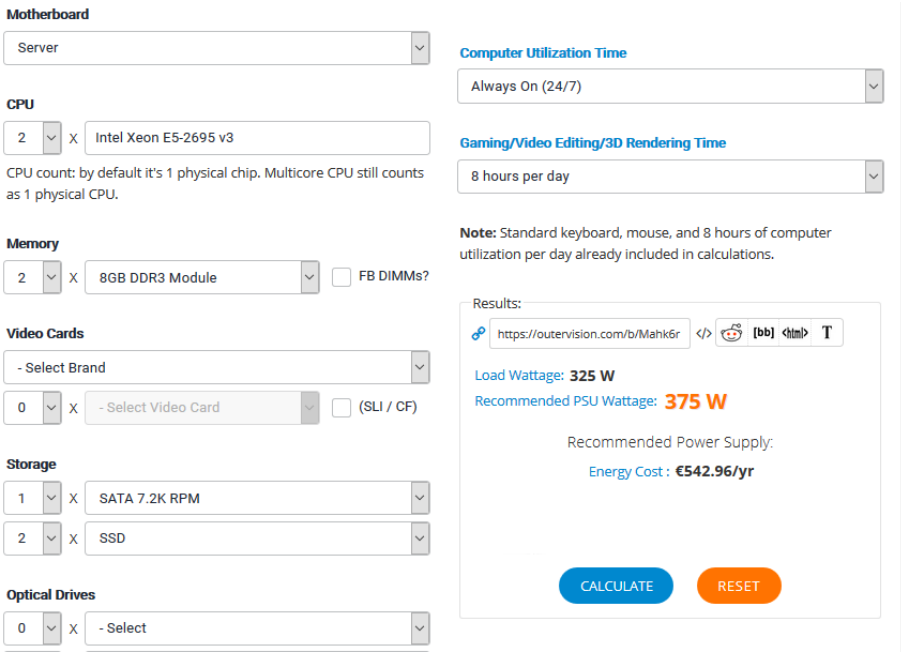


Figura 45: Captura del consumo estimado de la Workstation segun OuterVision© Power Supply Calculator

Según la Tabla 4, la potencia consumida por el *clúster* con 4 nodos es de **70,4W**. Por lo tanto, el consumo de la *Workstation* es **4,57 veces** mayor al consumo del *clúster* con 4 nodos. Recordemos que hemos medido el del *clúster* al alza y el de la *Workstation* a la baja.

En cuanto al coste de los componentes del ***clúster***, en nuestro caso particular con 4 placas DE1-SOC, pero suponiendo que contásemos con un *router* de 8 puertos y sin contar el coste del nodo maestro; ascendería a **776,04** €. Como se distingue en la Tabla 6.

NOTA: el precio de los componentes se ha calculado de forma aproximada. Las placas DE1-SOC tienen un coste de $250, pero $175 si es para universidades. En este caso se ha optado por el coste de estudiante $175. El cambio de dólares americanos a euros se ha hecho con el valor de cambio de divisa a fecha de escritura de este documento, que es de 1 USD = 0,891547682 EUR.

[13] https://ark.intel.com/es-es/products/81057/Intel-Xeon-Processor-E5-2695-v3-35M-Cache-2_30-GHz
[14] https://outervision.com/power-supply-calculator

	Precio/unidad (€)	Cantidad	Precio total (€)	TOTAL (€)
DE1-SOC[15]	156,02	4	624,08	**776,04**
Router 8 puertos Netgear	75	1	75	
Cable Ethernet	5	5	25	
Tarjetas de memoria micro SD	12,99	4	51,96	

Tabla 6: Coste del clúster con 4 nodos

El coste de la ***Workstation*** resulta más complicado de calcular, ya que no basta con sumar el precio de venta al público de cada uno de los componentes, puesto que estas estaciones de trabajo son vendidas por distribuidores. Según el *hardware* instalado en la *Workstation*, parece tratarse de un modelo de la serie 7047 de *Supermicro*[16]. Esta marca no presenta el precio de sus productos en su Web oficial, sino que nos insta a ponernos en contacto con sus distribuidores, cuyo nombre no se menciona por cuestiones publicitarias. No obstante, para obtener un precio aproximado que permita la comparación con el precio del *clúster,* se ha configurado un presupuesto partiendo del modelo *SuperWorkstation 7047A-T*[17] y un procesador algo peor en prestaciones (el utilizado no está disponible) y se ha obtenido un coste de 8503 dólares americanos, unos **7580,82 euros**, como se observa en la Figura 46.

Figura 46: Presupuesto Workstation, modelo 7047A-T

El coste de la *Workstation* es **10 veces** superior al coste del *clúster* con 4 nodos.

[15] http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=836
[16] https://www.supermicro.com/products/system/4U/7047/SYS-7047AX-TRF.cfm
[17] http://www.thinkmate.com/system/superworkstation-7047a-t

Por último, a modo de resumen final, la Tabla 7 presenta una comparativa entre el tiempo de ejecución (en el mejor de los casos: C u *OpenCL*) entre el *clúster* con 4 nodos y la *Workstation*, el consumo de energía cada programa en el *clúster* y en la *Workstation* y la relación entre dicho consumo. Para calcular el consumo de energía de cada programa (medido en kJ), se ha multiplicado el tiempo de ejecución correspondiente por el consumo de potencia del sistema en el que se ejecuta. Si la relación entre ambos consumos es menor que 1 significa que el *clúster* ha gastado menos energía que la *Workstation* en ejecutar el mismo programa con la misma entrada, y lo mismo en el caso contrario.

Programa	Entrada	Tiempo ejecución (seg): clúster (4 nodos)	Tiempo ejecución (seg): Workstation	Consumo de energía (KJ): clúster (4 nodos)	Consumo energía (kJ): Workstation	Energía: clúster / Workstation
Laplace	FullHD	5,38	1,10	0,37	0,35	1,06
	2k	6,42	1,35	0,45	0,43	1,05
	4k	10,22	2,34	0,71	0,75	0,95
	5k	11,96	3,30	0,84	1,06	0,79
	8k	29,49	4,73	2,07	4,52	0,46
Sobel	FullHD	5,63	1,24	0,39	0,39	1
	2k	6,65	1,64	0,46	0,52	0,88
	4k	10,34	2,79	0,72	0,89	0,81
	5k	12,14	3,73	0,85	1,20	0,71
	8k	29,41	5,32	2,07	1,71	1,21
Emborronamiento Gaussiano	FullHD	10,42	6,10	0,73	1,96	0,37
	2k	14,97	10,41	1,05	3,35	0,31
	4k	29,27	21,80	2,06	7,02	0,29
	5k	38,59	37,20	2,71	11,99	0,23
	8k	103,53	80,89	7,29	26,07	0,28
Flujo óptico	5	18,63	14,99	0,13	4,83	0,03
	10	37,74	30,47	2,65	9,82	0,27
	15	50,25	48,49	3,53	15,63	0,23
	20	64,04	63,79	4,51	20,56	0,22
	25	80,46	79,85	5,66	25,74	0,22
	30	95,83	94,98	6,75	30,61	0,22
	35	108,57	115,62	7,64	37,27	0,20
	40	121,59	129,09	8,56	41,61	0,21
	45	136,45	147,83	9,61	47,65	0,20
	50	154,59	162,47	10,88	52,37	0,21
Multifunction printer error diffusion	1	44,56	0,38	3,13	0,12	26,08
	4	71,18	2,14	5,01	0,68	7,37
	8	118,32	5,06	8,33	1,63	5,11
	12	166,21	8,42	11,70	2,71	4,32

Tabla 7: Comparativa entre el rendimiento, coste y consumo del clúster (4 nodos) frente a la Workstation

Como hemos visto en apartados anteriores, los tiempos de ejecución del *clúster* con 3 y 4 nodos son muy parecidos debido al cuello de botella causado por el uso de la red *Fast Ethernet*. Esto, junto a que el consumo del *clúster* con 3 nodos tiene un consumo significativamente menor que el caso anterior, con **49,08W** según los datos de la Tabla 4, ya que sólo es necesario uno de

los *routers*; se ha realizado esta misma comparativa con este caso. Estos resultados se recogen en la Tabla 8. Como se observa en ella, la eficiencia energética es mayor en el *clúster* con 3 nodos que en el caso con 4 en todos los programas. También vuelve a ser mejor con respecto a la *Workstation*, salvo en el programa *Multifunction printer error diffusion*.

Programa	Entrada	Tiempo ejecución (seg): clúster (3 nodos)	Tiempo ejecución (seg): Workstation	Consumo de energía (KJ): clúster (3 nodos)	Consumo energía (kJ): Workstation	Energía: clúster / Workstation
Laplace	FullHD	5,16	1,10	0,25	0,35	0,72
	2k	6,57	1,35	0,32	0,43	0,75
	4k	10,86	2,34	0,53	0,75	0,71
	5k	13	3,30	0,64	1,06	0,60
	8k	32,51	4,73	1,60	4,52	0,35
Sobel	FullHD	5,49	1,24	0,27	0,39	0,69
	2k	6,66	1,64	0,33	0,52	0,63
	4k	10,70	2,79	0,53	0,89	0,59
	5k	12,84	3,73	0,63	1,20	0,53
	8k	31,37	5,32	1,54	1,71	0,90
Emborronamiento Gaussiano	FullHD	11,87	6,10	0,58	1,96	0,30
	2k	18,13	10,41	0,89	3,35	0,27
	4k	36,85	21,80	1,81	7,02	0,26
	5k	47,66	37,20	2,34	11,99	0,20
	8k	135,18	80,89	6,63	26,07	0,25
Flujo óptico	5	24,62	14,99	1,21	4,83	0,25
	10	50,94	30,47	2,5	9,82	0,25
	15	56,71	48,49	2,78	15,63	0,18
	20	77,72	63,79	3,81	20,56	0,19
	25	89,59	79,85	4,4	25,74	0,17
	30	111,51	94,98	5,47	30,61	0,18
	35	128,11	115,62	6,29	37,27	0,17
	40	143,13	129,09	7,02	41,61	0,17
	45	158,08	147,83	7,76	47,65	0,16
	50	179,89	162,47	8,83	52,37	0,17
Multifunction printer error diffusion	1	44,44	0,38	2,18	0,12	18,18
	4	81,61	2,14	4,01	0,68	5,89
	8	122,44	5,06	6,01	1,63	3,69
	12	174,78	8,42	8,58	2,71	3,17

Tabla 8: Comparativa entre el rendimiento, coste y consumo del clúster (3 nodos) frente a la Workstation

5. Capítulo V: Conclusiones y trabajo futuro

5.1 Conclusiones y trabajo futuro

En este trabajo se ha presentado la creación de un clúster basado en FPGAs de bajo coste y consumo energético, con el fin de ejecutar programas de alta complejidad con un coste menor al de una *Workstation*. Lo más novedoso del clúster ha sido el uso de *OpenCL* para la implementación de los programas que se ejecutan en él, en combinación con el uso de placas DE1-SOC que cuentan con FPGAs de bajos recursos.

Se ha generado una imagen de UNIX/Linux para ser instalada en la DE1-SOC, basada en la última versión de *Debian* disponible, la cual es capaz de trabajar con las restricciones que hemos encontrado en las imágenes de UNIX/Linux disponibles en Internet, a saber: problemas de red relacionados con la configuración y/o el retardo de las comunicaciones y problemas con la ejecución de los programas escritos en OpenCL y compilados con el Intel FPGA SDK for OpenCL. Esta imagen está disponible públicamente en mi repositorio de mi Github [23].

Se han probado diversos *benchmarks* y se ha observado que el uso de *OpenCL* frente a *C* no siempre es mejor. Como hemos visto, el uso de *OpenCL* merece la pena cuando el tiempo de ejecución del código del *kernel* es muy inferior al tiempo necesario para copiar los datos de entrada y salida al espacio de memoria del dispositivo *OpenCL*. Esto puede conseguirse, por ejemplo, con el uso de un gran número de grupos de trabajo, aunque no todos los algoritmos son susceptibles de poder utilizarlos. Existen manuales con decenas de consejos sobre la optimización y el uso de recursos, pero cada uno de ellos depende del fabricante del dispositivo, como Intel, AMD o NVIDIA. Los tiempos obtenidos al ejecutar los programas de la suite en el clúster y en la *Workstation* han sido en su mayoría cercanos o incluso inferiores en el caso del clúster. En algunos casos el clúster ha sido directamente más rápido que la *Workstation*, mientras que en otros es cierto que, si bien el tiempo de ejecución de la *Workstation* ha sido menor, su consumo ha sido mucho más elevado que en el *clúster*. Además, el coste de la *Workstation* es 10 veces mayor que el del *clúster* propuesto con 4 nodos.

En cuanto a las limitaciones del sistema, hemos tenido problemas con los retardos en la red de conexión. Por cuestión de falta de recursos hemos utilizado *routers* de gama baja, los dados por las compañías telefónicas, cuyo rendimiento no es el adecuado para utilizarse en un clúster con las características que necesitábamos. Al igual que el rendimiento de cada nodo, contar con una red de altas prestaciones es de igual importancia para que el rendimiento de un clúster, se base en FPGAs o en otro hardware, no se vea condicionado por el cuello de botella que provoca.

Por último, consideramos que las siguientes tareas podrían realizarse como trabajo futuro:

- Generar imágenes de UNIX/Linux para la DE1-SOC con otros Sistemas Operativos, como *CentOS* y Ubuntu, para comparar su rendimiento con la imagen de *Debian* que se ha compilado para este proyecto y determinar la imagen que ofrece mayor rendimiento.
- Tratar de utilizar un *kernel* de Linux más moderno compatible con el *Intel FPGA SDK for OpenCL*, con el fin de utilizar un *kernel* más actualizado y, por lo tanto, con mejoras en la estabilidad, rendimiento, eficiencia y seguridad.

- Utilizar FPGAs con mayores recursos hardware o, incluso, crear un clúster heterogéneo utilizando placas con algunas FPGAs *high-end* y una mayoría de FPGA *low-end*; y repetir las comparaciones de los tiempos de ejecución, coste y consumo.
- Repetir las pruebas utilizándose una red Gigabit de altas prestaciones. Para ello sería necesario utilizar un único *router*, con cableado compatible con el estándar *Gigabit Ethernet* y una tarjeta controladora de red con esa misma tecnología en el nodo maestro.
- Añadir más programas a la suite para seguir evaluando el rendimiento y escalabilidad del clúster.
- Realizar las pruebas utilizando otros medios de almacenamiento como, por ejemplo, almacenamiento a través de la RED por medio del protocolo NFS.

5.2 Conclusions and Lines of future work

In this MSc Thesis a cluster based on low-end FPGAs has been presented in order to execute complex programs, requiring a lower power budget and cost than a workstation. The major novelty of the cluster is its ability to combine the low-end FPGAs with the mighty OpenCL parallel programming paradigm.

In order to solve the lack of suitable operating systems (OS) in literature, a UNIX/Linux image has been customized to be installed on the ARM Cortex-A9 hard cores that are embedded within the Altera DE1-SOC boards. This OS is based on Debian 8 and is able to deal with the restrictions found in the UNIX/Linux images available in literature: network problems related to configuration and/or communication delays and problems while running applications written and compiled in OpenCL using the Intel FPGA SDK for OpenCL. This UNIX/Linux image is public and available to download at my GitHub´s repository [23].

Several benchmarks have been tested and it has been observed that the use of OpenCL instead of C is not always better. As we have seen, the use of OpenCL is worthwhile when the kernel code execution time is much lower than the time needed to copy the input and output data to the OpenCL device memory. For example, this can be achieved by employing many working groups, although not all algorithms are able to leverage their use. There are manuals with tens of tips about resources usage and optimizations, but they depend on the device manufacturer, such as Intel, AMD or NVIDIA. The times obtained when running the benchmark suite in the cluster and in the Workstation have been mostly close or even lower in the case of the cluster. In some cases, the cluster has been directly faster than the Workstation, while in others the loss of performance is compensated by a lower energy consumption. In addition, the cost of the Workstation is around 10 times higher than the proposed cluster with 4 nodes.

Regarding the cluster limitations, we have had problems with network delays. Due to the lack of resources, we have used low-end routers supplied by the Internet Service Providers (ISPs), whose performance does not suit a high-performance cluster like ours. It is important to provide high-performance nodes, but is equally important, or even greater, to deploy a high-performance network to avoid an unnecessary bottleneck that prevents the nodes working at their maximum capacity.

Finally, we consider that the following tasks could be carried out as future work:

- Generating UNIX/Linux images for the DE1-SOC board based on another operating system, such as CentOS or Ubuntu, to compare their performance with the Debian image we have customized for this project and determine which is the image providing the highest performance.
- Trying to use a more modern UNIX/Linux kernel version compatible with the Intel FPGA SDK for OpenCL, with improvements in stability, performance, efficiency and security.
- Using FPGAs with more hardware resources or even creating a heterogeneous cluster based on boards with high-end and low-end FPGAs.
- Repeating the tests using a high-performance Gigabit Ethernet network. For this purpose, it is necessary to use a single router with Gigabit Ethernet protocol, as well as compatible wires and a master node with an installed Gigabit Ethernet network board.
- Adding more benchmarks to the suite to further evaluate the performance and scalability of the proposed cluster.
- Using another kind of storage media, such as network storage solutions through the NFS protocol.

6. Capítulo VI: Bibliografía

[1] D. James, «Moore's law continues into the 1x-nm era,» *2016 27th Annual SEMI Advanced Semiconductor Manufacturing Conference (ASMC),* 2016.

[2] T. Güneysu, C. Paar, J. Pelzl, G. Pteiffer, M. Schimmler y C. Schleiffer, «Parallel Computing with low-cost FPGAs. A framework for COPACOBANA,» [En línea]. Available: https://www.emsec.rub.de/media/crypto/veroeffentlichungen/2011/01/29/gpppss_parco07.pdf.

[3] J. Zhang y J. Li, «Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network,» *FPGA '17 Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays,* pp. 25-34, 2017.

[4] C. Rodriguez-Donate, G.Botella, C. García, E. Cabal-Yepez y M. Prieto-Matías, «Early Experiences with OpenCL on FPGAs: Convolution Case Study,» *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on,* 2015.

[5] Altera, «Using Linux on the DE1-SOC For Quartus Prime 16.0,» [En línea]. Available: ftp://ftp.altera.com/up/pub/Altera_Material/16.0/Tutorials/Linux.pdf.

[6] «The Angstrom Distribution,» [En línea].
Available: http://www.angstrom-distribution.org/.

[7] «About Yocto project,» 2016. [En línea]. Available: https://www.yoctoproject.org/about.

[8] Terasic, «DE1-SOC Board,» Terasic, 2013. [En línea]. Available: http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=205&No=836&PartNo=4.

[9] «Yocto Project. About OPKG,» 22 Mayo 2015. [En línea].
Available: https://git.yoctoproject.org/cgit/cgit.cgi/opkg/about/.

[10] «About U-Boot,» JaganTeki, 19 Noviembre 2013. [En línea]. Available: http://www.denx.de/wiki/U-Boot/Documentation.

[11] «Creating and Updating SD Card,» RocketBoards, [En línea]. Available: https://rocketboards.org/foswiki/view/Documentation/GSRD141SdCard.

[12] R. Bacrau, «GSRD - Boot Flow,» RocketBoards, 26 Junio 2014. [En línea]. Available: https://rocketboards.org/foswiki/view/Documentation/GSRDBootFlow.

[13] «Sitio oficial Linux Mint,» 2006. [En línea]. Available: https://www.linuxmint.com/.

[14] «Sitio oficial de Linaro,» 2010. [En línea]. Available: https://www.linaro.org/about/.

[15] «Repositiorio linux-socfpga,» [En línea]. Available:

https://github.com/altera-opensource/linux-socfpga.git.

[16] «Recompiling the Linux Kernel,» Intel Corporation, 8 Mayo 2017. [En línea]. Available: https://www.altera.com/documentation/ewa1403875738903.html#mwh1391806417857.

[17] «Sitio oficial Ubuntu,» [En línea]. Available: https://www.ubuntu.com/about.

[18] «Sitio oficial CentOS,» [En línea]. Available: https://www.centos.org/about/.

[19] A. A. D. B. G. B. y. C. G. D. G. Fernández, «Fast CU size decision based on temporal homogeneity detection,» de *2016 Conference on Design of Circuits and Integrated Systems (DCIS)*, Granada, 2016.

[20] «Intel FPGA SDK for OpenCL,» Intel Corporation, 8 Mayo 2017. [En línea]. Available: https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf.

[21] B. S. Morse, *Lecture 13: Edge detection,* Brigham Young University, 1998-2000.

[22] «Design Samples: Sobel filter,» Intel Corporation, [En línea]. Available: https://www.altera.com/support/support-resources/design-examples/design-software/opencl/sobel-filter.html.

[23] «tfm-de1soc-opencl-cluster,» [En línea]. Available: https://github.com/mariano2AA3/tfm-de1soc-opencl-cluster.

[24] G. Botella, A. García, M. Rodríguez-Alvárez, E. Ros, U. Meyer-Baese y M. C. Molina, «Robust Bioinspired Architecture for Optical-Flow Computation,» *IEEE Transactions on Very Large Scale Integration (VLSI) Systems ,* 2010.

[25] «Design Sample: Optical Flow,» Intel Corporation, [En línea]. Available: https://www.altera.com/support/support-resources/design-examples/design-software/opencl/optical-flow.html.

[26] J.-Y. v. Bouguet, «Pyramidal Implementation of the Lucas Kanade Feature Tracker Description of the algorithm,» *Intel Corporation.*

[27] D. Denisenko, «Lucas Kanade Optical Flow - from C to OpenCL on CV SoC,» 8 Julio 2014. [En línea]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/support/examples/download/exm_opencl_lucas_kanade_optical_flow_with_opencl.pdf.

[28] T. M. Hunter, D. D. S. K. J. M. Bold, P. T. y P. D. , «FPGA Acceleration of Multifunction Printer Image Processing using OpenCL».

[29] Altera, «Multifunction printer error difussion,» [En línea]. Available: https://www.altera.com/support/support-resources/design-examples/design-software/opencl/mfp-error-diffusion.html.

[30] S. Seo, J. Lee, G. Jo y J. Lee, «Automatic OpenCL work-group size selection for multicore CPUs».

[31] Micron, «TN-41-01: Calculating Memory System Power for DDR3,» 2007. [En línea]. Available: https://www.google.es/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwjO6ZPXx7DUAhUBVxoKHdIZAgYQFggmMAA&url=https%3A%2F%2Fwww.micron.com%2F~%2Fmedia%2Fdocuments%2Fproducts%2Ftechnical-note%2Fdram%2Ftn41_01ddr3_power.pdf&usg=AFQjCNF_AVCp.

[32] «The Angstrom Distribution,» 2016. [En línea]. Available: https://www.yoctoproject.org/product/angstrom-distribution.

[33] D. G. Fernández, A. A. D. Barrio, G. Botella, U. Meyer-Baese, A. Meyer-Baese y C. Grecos, «Pre-processing techniques to improve HEVC subjective quality,» *Proc. SPIE 10223, Real-Time Image and Video Processing 2017, 102230E,* 2017.

7. Capítulo VII: Anexos

7.1 Anexo I: código del lanzador de los algoritmos de detección de bordes con el operador de Sobel y Laplace y del emborronamiento gaussiano

```
#!/bin/bash

CONFIG="de1_soc-cluster.conf"     # Fichero de configuración nodos
RUNPATH="/tmp"                    # Directorio de trabajo en los nodos
THRESH=32                         # Umbral de Laplace
# THRESH=128                      # Umbral de Sobel


if [ $# != 5 ]; then
  echo "Uso: ./launcher_sobel-laplace_filter.sh [program] [kernel] [image]
[width] [height]"
  exit -1
fi

# Devuelve la hora actual en milisegundos
# Se utiliza para medir los tiempos
function get_time(){
  echo `date +%s%N`
}

# Ruta del nodo master donde se esta ejecutando este script
OUTPATH=$(pwd)

# Lee la configuracion del cluster
echo -n "Reading cluster config..."
IFS=$'\n' read -d '' -r -a NODES < $CONFIG
NUM_NODES=${#NODES[@]}
echo "(num nodes: $NUM_NODES)"

for ((i=0; i<$NUM_NODES; i++))
do
  echo " |- Node $i: ${NODES[i]}"
done
echo " |- done"


# Divide el argumento (una imagen) en NUM_NODOS trozos
echo "Spliting $3 in $NUM_NODES part(s)..."
time0=$(get_time)
WIDTH=$(($4/$NUM_NODES))
HEIGHT=$5
pamdice $3 -outstem=imageIn -width=$WIDTH -height=$HEIGHT
echo " |- done"
time1=$(get_time)

echo "Copying $1, $2 and $3 and executing..."
for ((i=0; i<$NUM_NODES; i++))
```

```
do
  echo -n " |- ${NODES[i]}:$RUNPATH ..."
  # Cambia el estado del nodo-i
  ssh root@${NODES[i]} 'echo "running" > state' &&
  ini=$(get_time) &&
  # Copia el programa, el kernel y el fragmento de imagen correspondiente
  scp  -q $1 $2 imageIn_00_0$i.ppm root@${NODES[i]}:$RUNPATH &&
  end=$(get_time) &&
  let SCPTIME[i]=$end-$ini &&
  echo -n " |- Time copying to node $i (sec): " &&
  echo "${SCPTIME[i]} * 0.000000001" | bc -l &&
  # Manda ejecutar el programa en el nodo-i llamando al script launch del nodo-i
 # Laplace y Sobel
  ssh root@${NODES[i]} "./launch.sh $OUTPATH/out $1 -img=imageIn_00_0$i.ppm
  -w=$WIDTH -h=$HEIGHT -thresh=THRESH" &
   # Emboronamiento Gaussiano
   # ssh root@${NODES[i]} "./launch.sh $OUTPATH/out $1 -input=imagen_00_0$i.ppm -w=$WIDTH -h=$HEIGHT -
output=out/out_imageIn_00_0$i.ppm -r=$6 -s=$7" &

  echo "done"
done
echo " |- done"
time2=$(get_time)


# Espera a que todas las placas finalicen y envien los resultados
echo "Waiting for termination..."
for ((i=0; i<$NUM_NODES;))
do
  state=$(ssh root@${NODES[i]} "cat state")
  if [ $state = "finish" ]; then
    echo " |- ${NODES[i]} finished"
    ssh root@${NODES[i]} 'echo "idle" > state' &
    i=$((i + 1))
  fi
done
echo " |- done"
time3=$(get_time)

# Une las imagenes de resultado
echo "Concating images..."
  pnmcat -leftright out/imageIn_00_0*.ppm > out/out_$6_$3
  rm out/imageIn_*.ppm
  rm imageIn_*.ppm
echo " |- done"
time4=$(get_time)



echo ""
echo "################################################"
echo ""
echo "                  EXECUTION TIME"
echo ""
```

```
let tiempo_total=$time4-$time0
let tiempo_dividir=$time1-$time0
let tiempo_ejecutar=$time3-$time1
let tiempo_juntar=$time4-$time3

echo -ne "  # Split input data (ms): \t\t"
echo "$tiempo_dividir * 0.000001" | bc -l

echo "  # Execution time (s):"

for ((i=0; i<$NUM_NODES; i++))
do
   node=$(ssh root@${NODES[i]} "cat /etc/hostname")
   nodeExecTime=$(ssh root@${NODES[i]} "head $1.log -n 1")
   nodeSCPTime=$(ssh root@${NODES[i]} "tail $1.log -n 1")
   echo -ne "     |- $node, execution time (sec): \t"
   echo "$nodeExecTime * 0.000000001" | bc -l
   echo -ne "     |- $node, SCP time (sec): \t"
   echo "$nodeSCPTime * 0.000000001" | bc -l
done
echo -ne "     |- TOTAL (sec):\t\t\t"
echo "$tiempo_ejecutar * 0.000000001" | bc -l


echo -ne "  # Join output (ms): \t\t\t"
echo "$tiempo_juntar * 0.000001" | bc -l

echo ""
echo "        ----------------------------- "
echo ""
echo -n "  TOTAL (s): "
echo "$tiempo_total * 0.000000001" | bc -l
echo ""
echo "################################################"
```

7.2 Anexo II: código del lanzador del algoritmo del Flujo óptico

```
#!/bin/bash

# ./launcher_optical_flow.sh optical_flow optical_flow.aocx input 3

if [ $# != 4 ]; then
  echo "Uso: ./launcher_image_filter.sh [program] [kernel] [images_prefix]
[N]"
  exit -1
fi

function get_time(){
  echo `date +%s%N`
}

# Variables
# Fichero de configuracion
CONFIG="de1_soc-cluster.conf"
# Ruta de trabajo en los nodos
RUNPATH="/tmp"
# Ruta donde se encuentran las imagenes de entrada
INPUTPATH="input/"
# Prefijo de las imagenes de entrada: frame0, frame1...frameN, por ejemplo.
INPUT=$3
# Extension de las imagenes de entrada
EXT=".raw"
# Número de imagenes a procesar
NUM_INPUT=$4
# Ruta del nodo master donde se esta ejecutando este script
OUTPATH=$(pwd)


# Lee la configuracion del cluster
echo -n "Reading cluster config..."
IFS=$'\n' read -d '' -r -a NODES < $CONFIG
NUM_NODES=${#NODES[@]}
echo "(num nodes: $NUM_NODES)"
for ((i=0; i<$NUM_NODES; i++))
do
  echo " |- Node $i: ${NODES[i]}"
done
echo " |- done"

# Procesa las imagenes de 2 en 2, mecanismo Round Robin
echo "Procesing images..."
time0=$(get_time)
for ((i=0, k=1; i<$NUM_INPUT-1; i++, k++))
do

  echo " |- Waiting for idle node to process $INPUTPATH$INPUT$i$EXT and
       $INPUTPATH$INPUT$k$EXT..."
```

```
  for ((j=0; j<$NUM_NODES;))
  do
    state=$(ssh root@${NODES[j]} "cat state")

    if [ $state = "idle" ] || [ $state = "finish" ]; then

         echo "   |- ${NODES[j]} ready. Copying input files and executing..."
         # Cambiamos el estado del nodo-j por running
         ssh root@${NODES[j]} 'echo "running" > state' &&

         # Copiamos el programa, el kernel y las imágenes al nodo-j
         scp -q $1 $2 $INPUTPATH$INPUT$i$EXT $INPUTPATH$INPUT$k$EXT
              root@${NODES[j]}:$RUNPATH &&

         # Mandamos ejecutar invocando al script launch en remoto
         ssh root@${NODES[j]} "./launch.sh $OUTPATH $1 -input1=$INPUT$i$EXT
             -input2=$INPUT$k$EXT" >> cluster_de1soc.log &
         # Retardo necesario para evitar que pregunte a la misma placa
         sleep 0.25
        break
     else
          j=$(((j + 1) % NUM_NODES))
     fi
done
done

# Esperamos a que todos los nodos finalicen
echo "Waiting for termination..."
for ((i=0; i<$NUM_NODES;))
do
  state=$(ssh root@${NODES[i]} "cat state")
  if [ $state = "finish" ]; then
    echo " |- ${NODES[i]} finished"
    ssh root@${NODES[i]} 'echo "idle" > state' &
    i=$((i + 1))
  fi
done
echo " |- done"

time1=$(get_time)

echo ""
echo "################################################"
echo ""
echo "                  EXECUTION TIME"
echo ""

let tiempo_total=$time1-$time0

echo ""
echo -n "  TOTAL (s): "
echo "$tiempo_total * 0.000000001" | bc -l
echo ""
echo "################################################"
exit 0
```