Facultad de Ciencias

Procesamiento de vídeo en dispositivos embebidos mediante OpenCL
(Video Processing on embedded systems using OpenCL)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Daniel Torre Miguel

Director: José Luis Bosque Orero

Co-Director: Raúl Nozal González

Junio - 2020

Tabla de contenido

Abstract:

This project explores the possibility of executing a program capable of doing real time video processing on a low computing power and low energy consumption heterogeneous system, which is composed by a CPU and a GPU. To achieve this, different programming models will be used such as programming using thread parallelism and host-device programming. To do this, tools like OpenMP and OpenCL were used.

Real time video processing is possible to do on the Raspberry Pi but, obtaining results such as 25 – 30 FPS is not possible due to the low computing capabilities of the system. OpenMP and OpenCL get similar performances about 15 FPS.

This platform can execute OpenCL kernels on CPU without issues, but to execute the same code on GPU there are several limitations that stem from the low compute power and experimental OpenCL implementation on the VideoCore IV. Nevertheless, OpenCL is portable, kernel code can be used on other systems without modifying it if the target system supports everything needed in the kernel. For this reason and its similar performance to OpenMP, it is recommended to use OpenCL.

Even if perfect results are not achieved, the platform is still interesting from a learning standpoint. It is a cheap and accessible system that can be used to learn parallel programming using shared-memory, distributed memory, or the host-device paradigm.

Moreover, it is still possible to explore this kind of system by using similar yet more powerful hardware or using other programming interfaces.

Resumen:

En este proyecto se explora la posibilidad de realizar procesamiento de vídeo en tiempo real en sistemas heterogéneos compuestos por una CPU y una GPU. Estos sistemas tienen una potencia de cómputo limitada, pero con la ventaja de tener un consumo de energía muy reducido. Para ello se medirá y optimizará el rendimiento de un programa desarrollado utilizando diferentes modelos de programación. Los modelos utilizados son el modelo de memoria compartida y el modelo anfitrión-dispositivo usando herramientas como OpenMP y OpenCL respectivamente.

Se ha encontrado que el procesamiento de vídeo en tiempo real es posible, aunque no obtiene resultados satisfactorios (25 - 30 FPS) dada la baja potencia del dispositivo utilizado. Utilizando tanto OpenCL como OpenMP se obtienen rendimientos similares, en torno a 15 FPS.

Se ha visto que la plataforma puede ejecutar *kernels* de OpenCL en CPU sin problemas, pero, para ejecutar el mismo código en GPU existen varias limitaciones debido a la limitada potencia de cálculo y la implementación experimental de OpenCL en Videocore IV. De todas formas, el código de los *kernels* de **OpenCL** es portable, por lo que **se puede ejecutar en otra plataforma sin necesidad de modificar el *kernel*** mientras tengan todo lo necesario para ejecutarlo. Por esta razón y su similitud de rendimiento con OpenMP, es conveniente utilizar OpenCL directamente.

Aunque no se obtengan los resultados esperados, la plataforma sigue siendo interesante desde el punto de vista de aprendizaje puesto que es un sistema barato, accesible y se puede utilizar para aprender a programar aplicaciones paralelas ya sea siguiendo el paradigma de memoria compartida, de memoria distribuida o el paradigma *host-device*.

Además, todavía es posible explorar este tipo de sistema en mayor profundidad ya sea utilizando otro hardware similar pero más potente o explorando otras interfaces de programación.

Palabras clave: Sistema embebido, arquitectura ARM, OpenCL, OpenMP

Capítulo 1: Introducción

En este capítulo se van a exponer las razones por las que se ha desarrollado este proyecto, las metas del desarrollo y que metodología se ha seguido en cada momento.

1.1 Sistemas heterogéneos de bajo consumo

El objetivo principal de este proyecto es comprobar cuanto rendimiento se puede extraer en un computador de baja potencia de cómputo y bajo consumo. Comprobando si se puede realizar procesamiento de vídeo en tiempo real.

Esto es interesante porque sistemas como la Raspberry Pi son baratos y accesibles además de ser útiles para diferentes usos. Además, al ser sistemas populares medir la viabilidad de implementar cuestiones de cómputo puede ser beneficioso para muchos usuarios. Aunque, es cierto que la variedad de modelos de Raspberry Pi con diferentes capacidades de cómputo puede ser un problema [1].

Lo que sí comparten todos los modelos de Raspberry Pi es su arquitectura heterogénea, todos los modelos utilizan procesadores Broadcom basados en ARM con una unidad de cómputo gráfico (GPU) integrada y cierta cantidad de Memoria RAM. Esta arquitectura no es exclusiva de la Raspberry Pi, existen otros sistemas como vim3, que persiguen más potencia de cómputo [2] o sistemas como Orange Pi [3] y Banana Pi [4] que persiguen eliminar carencias que tiene la Raspberry Pi como la ausencia de almacenamiento integrado o la baja potencia de la unidad de cómputo gráfico. Otros sistemas con similar o igual arquitectura son los populares Android TV que generalmente usan esta arquitectura salvo algunos casos particulares [5]. Pero, programar para estas plataformas es mucho más complejo debido a la dificultad de que supone cambiar el sistema operativo de la plataforma, de Android a Linux.

La existencia de plataformas con igual o similar arquitectura es favorable puesto que conlleva no tener que optimizar programas para la arquitectura del otro sistema, por ejemplo, los programas que utilizan la GPU como acelerador suelen requerir cambios para extraer el máximo rendimiento de diferentes sistemas y la carencia de drivers abiertos de OpenCL o portables a Linux.

En sistemas de procesamiento heterogéneos, (utilizan CPU y GPU, por ejemplo) es una necesidad adaptarse a la arquitectura, especialmente a la de la GPU puesto que esto implicará obtener rendimientos más altos. Estos sistemas utilizan un modelo de programación diferente conocido como *host-device* en los que la parte secuencial se ejecuta en la CPU, mientras que la parte intensiva en cómputo se descarga a la GPU. La ventaja de utilizar este modelo es la posibilidad de obtener mejoras de rendimiento muy grandes en ciertos tipos de problemas como procesamiento de imágenes.

Esto se debe a que las unidades gráficas están pensadas para cómputo gráfico y por lo tanto pueden realizar muchas operaciones sencillas de forma paralela. Esta capacidad se puede explotar en otros campos, por ejemplo, para cálculos físicos como dinámica molecular [6] o para acelerar procesos de inteligencia artificial como el *machine learning* [7].

Su principal desventaja es la complejidad de programación, aunque es cierto que existen herramientas como CUDA de NVIDIA que permiten ejecutar código de forma relativamente sencilla en una GPU. Pero, CUDA solo soporta dispositivos de NVIDIA, por lo que otros dispositivos necesitan herramientas diferentes. Por eso mismo, existen alternativas como OpenCL, un modelo de programación libre, soportado por una gran cantidad de fabricantes de dispositivos hardware que permite la programación portable de aplicaciones para sistemas heterogéneos.

Con OpenCL se puede generar código para sistemas heterogéneos de forma universal, utilicen GPU, unidades de procesamiento tensorial, matrices de puertas lógicas programables (FPGAs) u otros aceleradores hardware. Esto quiere decir que el código se puede utilizar en cualquier sistema heterogéneo independientemente de los dispositivos que tenga instalados. Es necesario que los drivers del sistema soporten OpenCL además de tener instalado el *runtime* y demás herramientas necesarias.

1.2 Procesamiento de vídeo en sistemas heterogéneos

El principal problema que se aborda en este Trabajo de Fin de Grado es la necesidad de optimizar el software para obtener resultados aceptables como una alta tasa de fotogramas por segundo en el procesamiento de vídeo en un sistema de bajas prestaciones y consumo energético.

El procesamiento de vídeo es un problema que necesita optimización puesto que trabaja con un número de datos relativamente grande, por ejemplo, una imagen sin comprimir en formato RGB de tamaño 1280x720 ocupa aproximadamente 2.7 Megabytes. Si el objetivo es procesar imágenes manteniendo una tasa de 30 fotogramas por segundo entonces hace falta procesar unos 81MB por segundo. Teniendo en cuenta además que muchos filtros tienen complejidades no lineales como el desenfoque gaussiano cuya duración crece de forma cúbica respecto al número de píxeles.

La ventaja es que es un problema con un alto grado de paralelismo en el que se pueden realizar muchas operaciones simultáneamente. Además, se dispone de hardware que permite computar de forma paralela operaciones sencillas, la GPU. Por eso mismo en este proyecto se utiliza OpenCL, aunque esto plantea nuevos problemas puesto que es un modelo de programación distinto y sensiblemente más completo.

Escribir código siguiendo el modelo de programación de dispositivo difiere bastante respecto a programar CPUs homogéneas, sobre todo debido a que ambos dispositivos disponen de espacios de memoria disjuntos:

- Las reservas de memoria de dispositivo se deben hacer previamente y de forma explícita, es decir, no basta con reservar memoria, sino que es necesario especificar que esa memoria se reserve en el dispositivo [8].
- También hace falta mover los datos de forma explícita para que el dispositivo los pueda utilizar.
- Es necesario evitar muchas estructuras de código típicas como las estructuras condicionales `if-else` ya que provocan pérdidas de rendimiento en estos sistemas [9] debido a la divergencia, que limita el grado de paralelismo al no poder paralelizarse la ejecución de los hilos que realizan la rama `if` con los que ejecutan la rama `else`.

Afortunadamente la librería utilizada, OpenCL, proporcionará funciones para hacer muchas de las tareas necesarias para ejecutar código en sistemas heterogéneos.

1.3 Objetivos

El objetivo principal es desarrollar una aplicación capaz de procesar vídeo en tiempo real con la máxima tasa de fotogramas por segundo posible en un dispositivo embebido y heterogéneo con poca capacidad de cómputo, pero con bajo consumo energético. Para ello este objetivo se ha dividido en los siguientes:

- Desarrollar un programa que lee una imagen y la procese. Esto conlleva aprender cómo están estructurados los datos en imágenes RGB, como trabajarlos y almacenarlos en disco.
- Paralelización del desenfoque gaussiano usando hilos del microprocesador (CPU) para mejorar el rendimiento.
- Mejorar la paralelización anterior cambiando del paradigma de memoria compartida al paradigma de *host-device*, utilizando la unidad de cómputo gráfico integrada (GPU) mediante el uso de OpenCL.
- Leer fotogramas de una cámara USB, necesario para tener obtener vídeo en tiempo real.
- Almacenar imágenes o fotogramas en un fichero en disco. Con ello se puede comprobar el correcto funcionamiento de los filtros. Además, se debe tener en cuenta este trabajo realizado por el programa cuando se apliquen optimizaciones puesto que generalmente es deseable tener el vídeo resultante en disco.
- Mostrar imágenes en pantalla. En algunas aplicaciones puede ser interesante ver los resultados en tiempo real por lo que esta funcionalidad también debe ser integrada para evaluar su rendimiento.

- Evaluar el rendimiento de los programas. Sin previa evaluación, optimizar un programa puede ser muy complejo puesto que se desconoce qué partes de este pueden estar ralentizándolo.
- Depuración y optimización, es importante experimentar con los programas desarrollados con el fin de asegurar que funcionan correctamente utilizando diferentes parámetros. Además, es necesario comprobar que funcionan sin fluctuaciones de rendimiento ya sea con los mismos parámetros o distintos.

Reutilizando el trabajo realizado en los subobjetivos, se codificó el programa principal utilizado para medir rendimientos y aplicar nuevas optimizaciones.

1.4 Metodología

1.4.1 Selección de plataforma

Inicialmente se consideró utilizar plataformas como Android TV y vim3, pero el trabajo necesario para obtener una distribución Linux y las herramientas necesarias funcionando era demasiado laborioso teniendo en cuenta la existencia de alternativas similares en potencia de cómputo y con mayor simplicidad.

Finalmente se usaron varias plataformas, principalmente una Raspberry Pi 3B con el procesador BCM2837 basado en diseños de ARM y un computador de sobremesa con un procesador AMD A10 7850k basado en la arquitectura X86.

Se decidió utilizar la Raspberry Pi 3B porque es extremadamente popular, existe mucha documentación sobre una gran variedad de temas, el procesador tiene múltiples núcleos y, por último, la facilidad de instalar Linux y las herramientas necesarias para desarrollar el proyecto. Además, esta plataforma ya se había utilizado con anterioridad para asegurar el funcionamiento de EngineCL en plataformas ARM. EngineCL es un *runtime* basado en OpenCL que simplifica la ejecución y balanceo de carga de *kernels* en sistemas heterogéneos [10]. Se valoró la utilización de EngineCL en este trabajo, pero requiere de un elevado conocimiento de OpenCL y C++ para poder adaptar los componentes de la arquitectura optimizada multi-hilo sin perjudicar el rendimiento.

1.4.2 Desarrollo del Proyecto

El esquema de trabajo seguido ha sido el siguiente: desarrollar código, comprobar su funcionamiento en todas las plataformas posibles, subir los cambios a un repositorio Git, realizar análisis de rendimiento de la aplicación, aplicar optimizaciones, comprobar el correcto funcionamiento de nuevo y por último realizar las mediciones y pruebas apropiadas para cada caso.

Con el fin de agilizar el desarrollo del proyecto, en la medida de lo posible se ha obtenido código de libre distribución desarrollado por otras personas de sus repositorios y páginas web públicas:

- Lectura de imágenes en formato de mapas de bits [11].
- Generar el módulo basado en Video 4 Linux [12].
- Documentación [13] y código fuente [14] de OpenCV, para implementar diferentes funcionalidades además de comprobar optimizaciones aplicadas en su código.

Se ha llevado un control de versión sobre todo el código desarrollado, las figuras y datos obtenidas en la experimentación y esta memoria en un repositorio Git privado.

1.5 Estructura del documento

En el capítulo 2 se expondrán las herramientas y algoritmos utilizados. Cada herramienta y algoritmo dispondrá de su propia sección detallando qué es, cuál es su funcionalidad y por qué ha sido incorporado o utilizado en este proyecto.

En el capítulo 2 explicará el desarrollo de la aplicación donde se explicarán los módulos generados, las optimizaciones aplicadas y algunas de las decisiones de diseño tomadas.

En el capítulo 4 se presentarán todos los experimentos y mediciones llevadas a cabo. Para cada experimento se explicará por qué se realiza ese experimento, qué resultados se esperaban, qué información nos aportan los resultados obtenidos y en caso de diferir, por qué no son similares o iguales a los resultados esperados.

Capítulo 2: Herramientas y algoritmos

En este capítulo se expondrán las herramientas y algoritmos importantes para el desarrollo de este proyecto.

Para una clara compresión del proyecto se presentarán tres secciones. La primera, las librerías de código utilizadas a lo largo del proyecto, la segunda los algoritmos de procesamiento utilizados y, por último, las herramientas y lenguajes.

2.1 Librerías

2.1.1 OpenCV

"OpenCV (Open Source Computer Vision Library) es una librería de visión artificial y aprendizaje automático. OpenCV fue creada con el objetivo de proveer una infraestructura estándar para aplicaciones de visión artificial y acelerar el uso de esta en productos comerciales" [15].

OpenCV es una librería de código abierto disponible en diferentes lenguajes como Python y C++. Esta librería facilita el procesamiento de video e imágenes proporcionando funciones para leer de disco o de un dispositivo como una webcam además de proporcionar algoritmos de procesamiento optimizados siguiendo diferentes estrategias.

Esta librería fue instalada para tener fácil acceso a la lectura de ficheros de vídeo en disco y usar cámaras web USB desde el código de la aplicación. Su simplicidad y amplia documentación fueron los principales factores que ayudaron a tomar esta decisión.

2.1.2 OpenCL

"OpenCL™ (Open Computing Language) es un estándar multiplataforma, abierto, libre de cánones, para la programación paralela de diversos procesadores encontrados en plataformas como computadores personales, servidores, dispositivos móviles y plataformas embebidas. OpenCL incrementa el rendimiento y velocidad de respuesta de un amplio espectro de aplicaciones en diferentes nichos de mercado como los videojuegos, el campo científico, software médico, herramientas artísticas profesionales, procesamiento de visión artificial y entrenamiento de redes neuronales" [16].

La característica más interesante que ofrece OpenCL es permitir la portabilidad de programas paralelos sin importar la configuración y dispositivos del sistema sobre el que se quiera ejecutar. Esto quiere decir que el mismo código se puede ejecutar tanto en CPU, GPU, Matrices de puertas lógicas programables (FPGA), Unidades de procesamiento tensorial (TPU) y otros dispositivos.

Estos requisitos de portabilidad fuerzan a OpenCL a tener un proceso de configuración, compilación y ejecución lo bastante genéricos para poder generar código máquina de todos los dispositivos. Esto causa que OpenCL sea muy verboso y complejo si se compara con otras soluciones como puede ser CUDA de NVIDIA.

La principal desventaja de este conjunto de librería y *runtime* es tener que buscar y reconocer dispositivos de forma dinámica para tener código paralelo portable. Esto conlleva no poder tener en el propio código la plataforma y dispositivo objetivos requeridas para compilar el código.

Para ello OpenCL proporciona multitud de funciones que permiten esta configuración además de la compilación del código específico para cada dispositivo en tiempo de ejecución del programa.

En este proyecto OpenCL se utiliza principalmente para poder ejecutar los filtros de imágenes de forma paralela sobre GPU en las plataformas utilizadas. También se utiliza para su rendimiento de CPU frente a otras soluciones como OpenMP.

Como ejemplo de programación de *kernel* se va a usar el siguiente bloque de código:

```
1. int i;
2. float a[10], b[10], c[10];
3. for(i = 0; i< 10; i++)
4. {
5.    c[i] = a[i] + b[i]
6. }
```

En este código simplemente se suman dos vectores de tamaño 10 y el resultado se guarda en otro vector.

Ahora se expondrá su equivalente en OpenCL:

```
1. __kernel void vectorSum(__global float *a, __global float *b, __global float *c,
2.                                    int size)
3. {
4.    int index = get_global_id(0); //get thread id
5.    if(index < size)
6.    {
7.       c[index] = a[index] + b[index];
8.    }
9. }
```

Para definir una función de *kernel* se debe utilizar la palabra reservada `__kernel`. La otra palabra reservada importante en este código es `__global` que se utiliza para decir que una variable está en la zona de memoria global del dispositivo.

En el código lo que ocurre en este caso es que cada instancia del kernel obtiene su id y lo utiliza como índice para sumar los vectores. Se deben generar un número de instancias de kernel igual al número de elementos del vector. Si se generasen más instancias de las debidas, la comprobación `index < size` se encarga de que no ocurra ningún error.

2.1.3 OpenMP

"La misión de la ARB (Architecture Review Boards) de OpenMP es estandarizar el paralelismo multi-lenguaje basado en directivas puesto que es productivo, portable y ofrece buen rendimiento.

Definida por la unión de un grupo de las mayores empresas de hardware y software, la API de OpenMP es portable, con un modelo escalable que proporciona a los programadores que utilizan el modelo de memoria compartida una interfaz sencilla y flexible para desarrollar aplicaciones paralelas en una gama de plataformas, desde sistemas embebidos y dispositivos aceleradores a sistemas multinúcleo con memoria compartida. La ARB de OpenMP es dueña de la marca OpenMP, supervisando su especificación y produciendo y aprobando nuevas versiones de esta." [17].

OpenMP es una combinación de funciones, directivas de compilador y un entorno de ejecución que permiten la fácil extracción de paralelismo en lenguajes como C/C++ y Fortran.

OpenMP utiliza únicamente el modelo de memoria compartida dónde varios hilos de CPU comparten memoria lo que permite que estos hilos accedan a datos comunes y puedan comunicarse entre sí. Esto no implica que cada hilo no pueda definir datos de forma privada.

Aunque no sea tan versátil y potente como las librerías estándar de hilos de Linux, permite que el código sea limpio y portable sin perjudicar al rendimiento obtenido.

En este proyecto OpenMP se usa para optimizar los filtros utilizando paralelismo de datos o paralelismo de tareas dependiendo de los parámetros utilizados en tiempo de compilación.

En el siguiente bloque de código podemos observar cómo extraer paralelismo de datos utilizando las directivas de OpenMP [18]:

```
1.  void parallelAddition (unsigned N, const double *A, const double *B, double *C)
2.  {
3.     unsigned i;
4.     #pragma omp parallel for shared (A,B,C,N) private(i) schedule(static)
5.     for (i = 0; i < N; ++i)
6.     {
7.        C[i] = A[i] + B[i];
8.     }
9.  }
```

La cláusula `parallel` provoca la creación de un número de hilos decidido por el entorno de OpenMP en tiempo de ejecución (se puede modificar este número usando variables de entorno, la cláusula `num_threads` o utilizando la función `set_num_threads`, la cláusula `for` provoca que el compilador utilizando las herramientas de OpenMP paralelice ese bucle dividiendo las iteraciones entre los hilos, estos hilos ejecutan el mismo código utilizando diferentes datos. Las cláusulas `shared` y `private` le indican al compilador como son las variables, si compartidas entre hilos o privadas respectivamente. Si son privadas entonces cada hilo tiene su propia copia mientras que si es compartida entonces hay una única copia a la que todos los hilos pueden acceder en cualquier momento. Por último, nos encontramos la cláusula `Schedule` que modifica como se reparte y ejecuta el trabajo dependiendo del valor utilizado:

- Para `static` el reparto de la carga es equitativo y completo, es decir, el número de iteraciones se divide entre el número de hilos. Este reparto de trabajo es inmediato y sencillo, aunque conlleva una pequeña pérdida de rendimiento si algún hilo se retrasa.
- Para `dynamic` se generan bloques de trabajo más pequeños (configurable) que se reparten de forma dinámica, cuando un hilo termina de computar el bloque asignado se le asigna otro. Esto pretende evitar la pérdida de rendimiento que se puede producir si se utiliza el planificador estático.
- Por último, `guided` es un planificador hibrido que divide el trabajo de forma similar al planificador dinámico, pero utiliza bloques de tamaño variable, empieza con bloques grandes y a medida que se agotan se genera bloques más pequeños. Este planificador pretende resolver el problema que tiene el planificador estático gastando menos tiempo en repartir y asignar el trabajo que el planificador dinámico [19].

Como ejemplo de extracción de paralelismo de tareas el código anteriormente mostrado se ha modificado:

```
1.  void parallelAddition(unsigned N, const double *A, const double *B, double *C)
2.  {
3.     unsigned i;
4.     #pragma omp parallel shared(A, B, C, N) private(i)
5.     {
6.        for (i = 0; i < N; ++i)
7.        {
8.           #pragma omp task
9.           {
10.             C[i] = A[i] + B[i];
11.          }
12.       }
13.    #pragma omp taskwait
14.    }
15. }
```

En este código, se considera que cada suma es una tarea que se genera usando la cláusula `task` y al final del bucle se espera a la finalización de todas las tareas usando la cláusula `taskwait`. El funcionamiento de este código es muy similar al caso anterior, pero usando el planificador dinámico con un tamaño de bloque de una iteración.

En este caso no se utiliza el modelo de tareas de forma óptima puesto que su objetivo es explotar el paralelismo de tareas donde varios hilos ejecutan diferente código diferente de forma simultánea.

2.1.4 Video 4 Linux 2

Video 4 Linux 2 (V4L2) es la segunda versión de Video 4 Linux, una colección de drivers y una Interfaz de Programación de Aplicaciones (API). Estas permiten la estandarización de los dispositivos de vídeo bajo Linux además de facilitar al programador el acceso a los dispositivos de vídeo.

V4L2 se escogió para reemplazar OpenCV porque su rendimiento no era suficientemente bueno. También se escogió porque permite cambiar ciertos parámetros como el tamaño y el espacio de color de forma más sencilla.

En este proyecto V4L2 se utiliza únicamente para extraer fotogramas de dispositivos de vídeo como webcams.

2.1.5 Seaborn

"Seaborn es una librería de visualización de datos para Python basada en MatPlotLib aportando una interfaz de alto nivel para dibujar figuras de estadísticas de forma atractiva e informativa." [20].

Seaborn es usado en este proyecto para representar datos obtenidos del programa principal desarrollado, principalmente datos de rendimiento como pueden ser los fotogramas por segundo.

Tiene muchas formas de representar datos sin necesidad de configuraciones complejas además de generar gráficas buenas con la configuración por defecto. Por esto mismo se utilizó Seaborn en vez de otras librerías más comunes como MatPlotLib.

Además, es compatible con Pandas, la librería utilizada para acceder a los datos generados por la aplicación principal mientras que para utilizar MatPlotLib se necesitaban adaptar los datos.

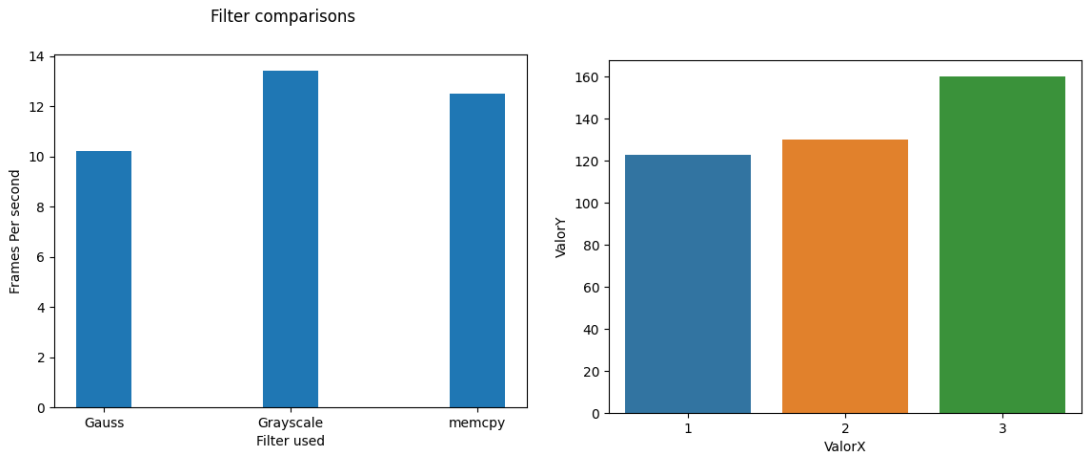

Figura 2.1: Comparación entre Matplotlib (izquierda) y seaborn(derecha)

En la *Figura 2.1* podemos ver a la izquierda el resultado de utilizar MatPlotLib con configuración mínima teniendo que adaptar los datos frente a Seaborn en la derecha, que con configuración mínima genera un resultado mejor sin necesidad de tener que adaptar los datos.

2.1.6 Pandas

"Pandas es una librería para Python de código abierto rápida, potente, flexible y fácil de usar para manipular y analizar datos." [21].

Para trabajar con los datos generados se utilizará esta librería que es muy útil puesto que permite trabajar con datos en formato de Valores Separados por Comas (CSV) que es el formato utilizado.

La principal ventaja de usar Pandas es la posibilidad de usar sus estructuras en conjunto con Seaborn para representar los datos de forma sencilla y rápida sin necesidad de transformarlos.

Por estos dos motivos Pandas fue escogido en vez de otras alternativas.

2.2 Algoritmos

2.2.1 Desenfoque gaussiano

El desenfoque gaussiano es un algoritmo de tratamiento de imágenes que toma una imagen inicial y genera una versión borrosa o desenfocada [22].

Específicamente en este proyecto se usa de la siguiente forma:

$$p_i = \sum_{j=3}^{n=-3} X_{i-j} \cdot Y_i$$

Básicamente, el valor de cada píxel i de la imagen de salida de calcula como el producto escalar de dos vectores, un vector que es una parte de la imagen de entrada y otro que es la función gaussiana ambos del mismo tamaño y con el píxel i como elemento central del vector.

Este algoritmo fue escogido como filtro es utilizado con frecuencia para reducir el ruido indeseado en una imagen. Si los sistemas utilizados no logran buen rendimiento con este algoritmo entonces no lograrán buen rendimiento en algunos casos de procesamiento de vídeo de tiempo real.

Por ejemplo, la detección de aristas utiliza el desenfoque para reducir ruido:

En la *Figura 2.2* vemos cuatro resultados de un algoritmo de detección de aristas, el primer resultado se obtiene procesando la imagen original y los sucesivos resultados se obtienen procesando antes la imagen inicial usando el desenfoque gaussiano. Los sucesivos resultados han sido sujetos a un desenfoque más fuerte que causa que la imagen sea más borrosa.

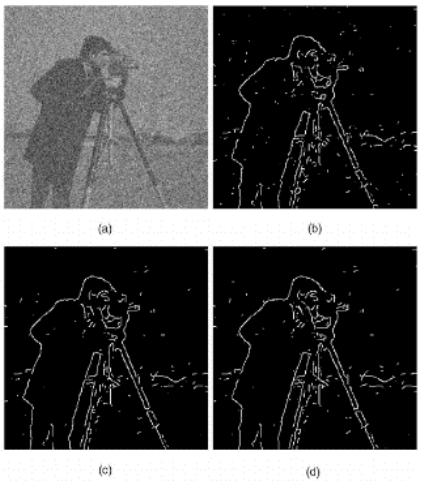

Figura 2.2: Detección de aristas en imágenes filtradas con desenfoque Gaussiano

2.2.2 Conversión a escala de grises

Este algoritmo convierte una imagen en formato RGB a una imagen en escala de grises [23].

Para llevar a cabo esta transformación es necesario calcular una media ponderada de los valores de los colores de cada píxel.

Para cada píxel p_{ij} su valor resultante g se obtiene como:

$$g = 0.3 \cdot rojo_{ij} + 0.59 \cdot verde_{ij} + 0.11 \cdot azul_{ij}$$

Estos valores no son arbitrarios, se han escogido de acuerdo con las longitudes de onda de cada color y el brillo que emiten.

Este algoritmo se ha escogido porque también es útil para el procesamiento de vídeo puesto que utilizando la escala de grises se puede detectar las diferencias en brillo en una imagen de forma sencilla. Esto permite, por ejemplo, detectar rostros.

Además, su baja complejidad, O(n), lo convierte en un algoritmo interesante para comparar el comportamiento entre CPU y GPU cuando el trabajo a realizar no es muy intenso.

2.2.3 Warp Perspective

Este algoritmo toma una imagen y una matriz 3x3 de transformación como entrada y obtiene una imagen con una perspectiva diferente. La perspectiva obtenida depende de la matriz de transformación utilizada.

Para realizar esta transformación se puede utilizar la siguiente función [24]:

$$Imágen\ de\ destino(x,y)\ =\ fuente\left(\frac{M_{11}x + M_{12}y + M_{13}}{M_{31}x + M_{32}y + M_{33}}, \frac{M_{21}x + M_{22}y + M_{23}}{M_{31}x + M_{32}y + M_{33}}\right)$$

Donde fuente es la imagen original, *x* e *y* son las coordenadas del píxel actual y M_{xy} el valor en la posición *xy* de la matriz de transformación.

Esta función lo que hace es obtener el píxel *xy* de la imagen de destino a partir de un píxel en la imagen fuente. Las coordenadas de este píxel se obtienen aplicando la fórmula que se ha enseñado anteriormente

Figura 2.3: Ejemplo de resultado obtenido

En la figura anterior [25] se puede observar un resultado obtenido por este algoritmo. A la izquierda se encuentra una fotografía de un formulario y a la derecha la misma fotografía transformada para que solo se vea el formulario en la perspectiva deseada.

2.3 Otras herramientas

2.3.1 C/C++

Inicialmente el trabajo se comenzó a desarrollar en C porque OpenMP y OpenCL ofrecen soporte para este lenguaje. Además, utilizar lenguajes compilados ayuda a explotar el máximo rendimiento posible.

El cambio de C a C++ fue necesario puesto que las versiones más recientes de OpenCV ya no soportan C. Además, la librería de estándar de hilos de C++ es más sencilla de usar y facilita algunas tareas realizadas en el proyecto.

2.3.3 FFMPEG

“Una solución completa, multiplataforma para grabar, convertir y transmitir audio y vídeo” [26].

FFmpeg es una herramienta multiplataforma para grabar y convertir flujos de vídeo y audio. Permite configurar una amplia variedad de parámetros como el tamaño, tasa de fotogramas, el espacio de color, códec de salida y más. En este proyecto se ha utilizado para comparar rendimientos de grabación de vídeo con las librerías utilizadas en el programa principal.

Capítulo 3: Procesamiento de imágenes en sistemas heterogéneos de bajo coste

En este capítulo se presenta todo el trabajo de desarrollo realizado a lo largo del proyecto, dividido en una serie de etapas.

Como paso previo fue necesario comenzar con la configuración e instalación de las herramientas necesarias, librerías, entornos de ventanas, configuración de DNS entre otros.

El desarrollo comienza con un programa en el que se lee una imagen en formato de Mapa de Bits y se filtra con el desenfoque gaussiano. Posteriormente el programa se modifica para utilizar OpenCL con el objetivo de familiarizarse con la generación de *kernels* y su ejecución.

En tercer lugar, se generan programas de prueba para utilizar OpenCV y V4L2, obteniendo fotogramas de webcams y ficheros, guardándolos en disco y mostrándolos en pantalla.

Seguidamente se utiliza el código desarrollado para filtrar imágenes y leer fotogramas para generar un programa secuencial que procesa vídeo de webcams o ficheros.

Por último, se toma este programa y se le aplican todas las optimizaciones necesarias.

Respecto a la configuración del sistema, por un lado, se utiliza una Raspberry Pi 3B con una webcam Netway NW798 y por otro un computador x86 con una webcam Logitech C920. Ambas webcams soportan un tamaño de vídeo máximo de 1280x720 con una tasa de fotogramas por segundo de 30.

3.1 Filtrar una imagen

Este programa de procesamiento de imágenes simplemente recibe por parámetro una imagen en formato de Mapa de Bits (BMP) y le aplica el filtro de desenfoque gaussiano guardando el resultado en un fichero distinto.

El código está distribuido en diferentes módulos C.

1. Gestión de imágenes BMP
2. Programa principal y filtro de desenfoque
3. Lanzador de *kernels* de OpenCL

3.1.1 Módulo BMP

El módulo que gestiona las imágenes BMP contiene las estructuras de datos y funciones necesarias para leer y guardar imágenes en formato BMP con 24 bits por píxel (bpp) además de tener una estructura que agrupa los tres canales de color rojo, azul y verde para acceder a ellos sin tener que calcular desplazamientos.

Los ficheros tienen la siguiente estructura [27]:

- Un campo numérico llamado tipo que debe ser arbitrariamente 0x4D42
- Una cabecera de fichero que debe contener el tamaño, 4 bytes reservados y un número que contenga el desplazamiento hasta los datos en bytes.
- Otra cabecera que contiene la información importante de la imagen como el tamaño, los bpp, colores…
- Los datos están estructurados como un vector de N * 3 bytes, cada tres bytes es un píxel siguiendo un orden de rojo, verde y azul.

El módulo solo tiene dos funciones, una para leer imágenes y otra para guardarlas.

Para obtener el contenido de una imagen BMP se debe abrir el fichero y usando la información de la cabecera de fichero se obtiene un puntero a los datos y con la segunda cabecera obtenemos el tamaño de los datos. Con estos dos datos se pueden leer todos los pixeles de la imagen.

Para guardar la imagen en disco solo hay que seguir el proceso inverso, con el tamaño y datos de la imagen generar ambas cabeceras y escribir a disco siguiendo la estructura.

3.1.2 Modulo principal

Este módulo contiene el programa principal y las funciones necesarias para aplicar el desenfoque gaussiano.

Utilizando el módulo anterior el programa carga una imagen en memoria y obtiene un vector con cierto número de estructuras de tipo píxel. Después genera una matriz con el filtro o función gaussiana de desenfoque. Seguidamente le aplica esta función gaussiana a la imagen de entrada y por último guarda la imagen resultante en memoria.

Como se ha mencionado en la introducción de la Sección 3, este programa tiene dos versiones: una secuencial que acaba de explicarse y otra paralela utilizando OpenCL.

3.1.3 Modulo para lanzar *kernels* de OpenCL

Para lanzar *kernels* de OpenCL en el software desarrollado en este proyecto se ha programado un módulo con las funciones necesarias para seleccionar dispositivo, compilar el *kernel*, gestionar los buffers y ejecutarlo.

El proceso de ejecutar un kernel de OpenCL requiere varios pasos previos:

1. Reconocimiento del sistema, esto significa buscar las plataformas y sus dispositivos que tengan un controlador compatible con OpenCL instalado. Para ello existen las funciones `clGetPlaformInfo` y `clGetDeviceInfo`.
2. Creación del contexto, el usuario escoge la plataforma y dispositivo deseados y con ambas se crea el contexto requerido por OpenCL para compilar y ejecutar el programa usando `clCreateContext`.
3. Crear la cola de comandos, estas colas contienen las instrucciones para informar cuál de los dispositivos del contexto a utilizar (puede haber varios por contexto) y cómo lo debe ejecutar (en orden o fuera de orden). Para crear la cola existe la función `clCreateCommandQueue`.
4. Usando la función `clCreateProgramWithSource` se compila el programa proporcionándole a esta función la cadena de texto, donde se encuentra el código del *kernel*, y el contexto generado anteriormente. Posteriormente se debe llamar a `clBuildProgram` para finalizar la generación del binario a ejecutar.
5. Creación de buffers de entrada y salida. Para enviar y recibir grandes cantidades de datos como arrays, se deben usar buffers de entrada y salida. Estos se pueden crear usando `clCreateBuffer`. Se pueden configurar de diferentes formas, pero principalmente se debe utilizar la directiva `CL_MEM_READ_ONLY` con los buffers de entrada y `CL_MEM_WRITE_ONLY` con los de salida.
6. El siguiente paso es preparar los argumentos de la función del *kernel* a ejecutar. Para ello se utiliza la función `clSetkernelArg` a la cual se le pasa como parámetros, los valores numéricos de aquellos parámetros que sean simplemente valores sueltos y la dirección de los buffers de entrada y salida en caso de ser estructuras de datos.
7. El último paso antes de lanzar el comando de ejecución del *kernel* es encolar una orden de escritura de datos utilizando `clEnqueueWriteBuffer`.
8. Finalmente, para ejecutar el *kernel*, se utiliza la función `clEnqueueNDRangeKernel` a la cual se le deben de pasar por parámetro el tamaño global (número de *work-items*) y el tamaño local, que es el tamaño de los grupos de *work-items* llamados *work-groups*. Esto se explicará más adelante, pero lo más importante es que el tamaño global determina el número de iteraciones que se deben ejecutar en total por los *work-items* que se generan. El número de *work-items* generados depende de la arquitectura del dispositivo y la implementación de OpenCL en este.

9. Para terminar, usando un parámetro de retorno de la función anterior, se espera a la finalización del *kernel* usando `clWaitForEvents` y posteriormente se recogen los datos de salida usando `clEnqueueReadBuffer` que envía una petición de lectura de datos.

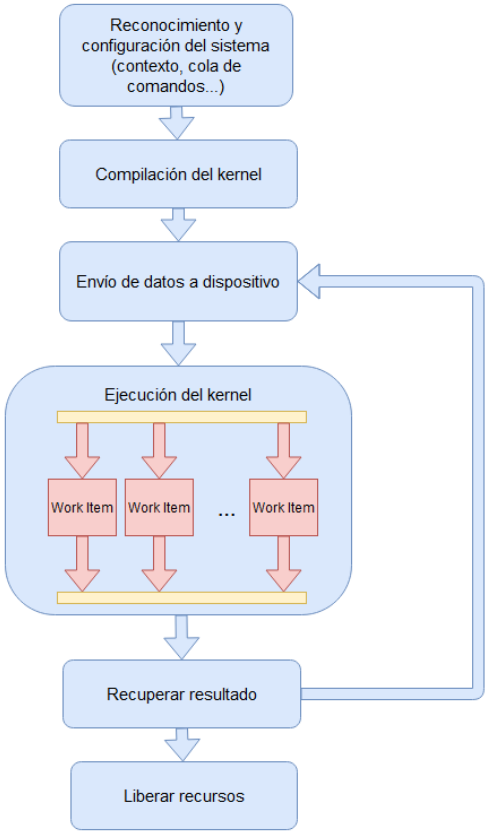

Figura 3.1: Proceso de ejecución de kernel de OpenCL

En la *Figura 3.1* se puede observar el proceso de ejecución de un *kernel* de forma resumida. También se puede ver que OpenCL permite reutilizar *kernels*. Una vez finalizada la ejecución se pueden cargar datos de entrada nuevos y volver a ejecutar con los mismos parámetros.

Para obtener el máximo rendimiento estos pasos no son los más apropiados. Son los que habría que dar inicialmente para obtener una primera aproximación al rendimiento deseado.

Uno de los principales pasos a modificar es el lanzamiento del *kernel*. Este paso se puede optimizar utilizando el tamaño local o tamaño de *work-group* adecuado para adaptarse a la arquitectura. Para ello conviene conocer cómo funcionan los *kernels* y los tamaños de *work-items* y *work-groups* con cierta profundidad.

Los *kernels* de OpenCL pueden ser multidimensionales, esto quiere decir que los *work-items* y *work groups* se pueden organizar en estructuras n-dimensionales. Cada *work-item* entonces tiene varios identificadores, uno por cada dimensión del kernel. Esto permite adaptarse mejor a los datos del problema que se intente resolver, por ejemplo, en un kernel bidimensional, se puede tratar una imagen como una matriz. Entonces, cada *work-item* accede a su píxel correspondiente utilizando su identificador "x" y su identificador "y".

Esto también se aplican a los *work-groups*, cada *work-item* tiene sus identificadores dentro del *work-group*.

En la siguiente ilustración se puede ver lo anteriormente descrito de forma gráfica [28].

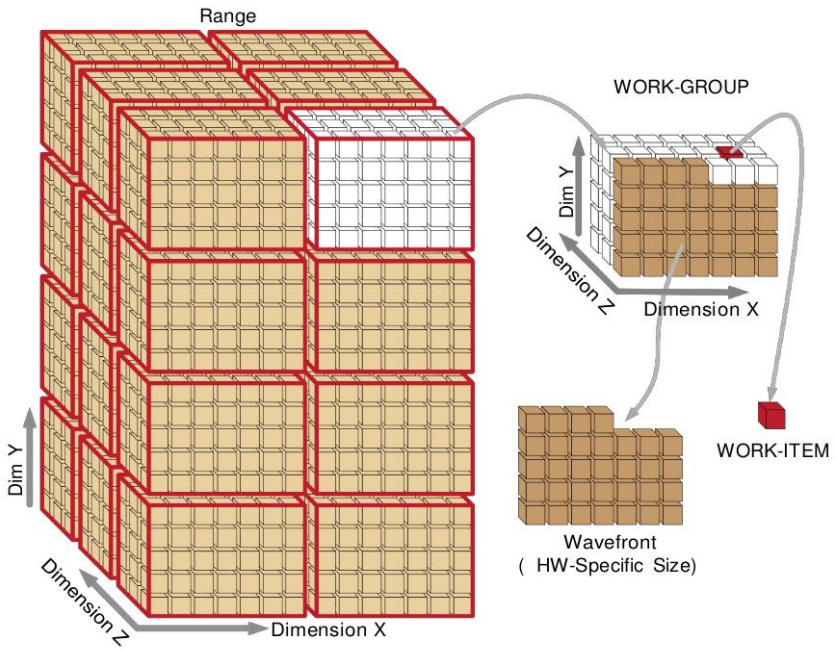

Figura 3.2: Representación gráfica de work-items y work-groups

Ejemplo de lanzamiento de kernel bidimensional:

```
1.  size_t global[2] = {640, 480};
2.  size_t local[2] = {16,  32};
3.  clEnqueueNDRangeKernel(ocl_program->queue,
4.                                  ocl_program >kernel, 2, NULL, global, local, 0, NULL,
5.                                  &kernel_event);
```

En el código anterior se generan 640x480 *work-items* divididos en grupos de tamaño 16x32.

Para obtener estos identificadores se usa la función `get_global_id(dim)` en el código del *kernel*, donde `dim` es el número de la dimensión de la cual se quiere obtener el identificador global. Para sus identificadores dentro de su grupo se usa `get_local_id(dim)`.

Como se ha visto anteriormente en el código, el número total de *work-items* y el tamaño de los *work-groups* se le pasa a la función `clEnqueueNDRangeKernel` como tuplas.

Cada tupla puede tener varios valores, pero ambas tuplas (global y local) deben tener el mismo tamaño para el correcto funcionamiento del kernel puesto que el tamaño de estas se le debe pasar por parámetro.

Además de facilitar la programación, las divisiones en dimensiones y grupos de OpenCL permiten también que los *work-items* compartan memoria y barreras de ejecución dentro del mismo *work-group*.

Entonces, escoger los parámetros adecuados permite adaptarse mejor a la arquitectura subyacente del dispositivo puesto que OpenCL podrá asignar *work-items* y *work-groups* para aprovechar las unidades de cómputo al máximo. Elegir un tamaño demasiado pequeño implica llenar todas las unidades de cómputo sin aprovecharlas al 100% de su capacidad y teniendo que esperar a que haya más unidades libres mientras que elegir un tamaño demasiado grande implica sobrecargar a un grupo de unidades y tener que esperar a que estas completen todo el trabajo teniendo otras unidades libres.

3.2 Obtención de vídeo

Para poder hacer procesamiento de vídeo en tiempo real se necesita extraer fotogramas de dispositivos de vídeo desde el programa. En este proyecto se han utilizado varias cámaras web o webcams.

Para obtener vídeo de la webcam se han utilizado dos librerías distintas, inicialmente OpenCV y posteriormente Video 4 Linux 2. La ventaja de OpenCV es que obtiene vídeo de forma sencilla, aunque algunos parámetros son complejos de cambiar como los espacios de color. La ventaja de Video 4 Linux es la capacidad de configurar todos los parámetros posibles.

Se cambió de OpenCV a Video4Linux porque inicialmente se obtuvieron resultados poco satisfactorios en la obtención de fotogramas. Con este cambió se intentó obtener una mejor tasa de fotogramas por segundo.

De todas formas, ambos funcionan de forma similar, siguiendo el siguiente proceso:

- Configuración de la cámara, apertura de conexión y fijación de parámetros.
- Recogida de fotogramas, se pueden obtener durante un tiempo indefinido.
- Cierre de conexión y liberación de recursos.

Para la versión de V4L2 se programó un módulo con la funcionalidad anteriormente descrita. A diferencia de OpenCV, es necesario utilizar peticiones usando una función llamada `v4l2_ctl`.

Para abrir la conexión con la cámara simplemente hay que abrir el correspondiente fichero de vídeo en la carpeta "dev" en la raíz del disco utilizando la función `v4l2_open`. Generalmente con solo una cámara conectada, será `/dev/video0`. El siguiente paso es configurar parámetros como el tamaño y el espacio de color en las estructuras de V4L2 y enviar una petición `VIDIOC_REQBUFS`.

El último paso para finalizar la inicialización es reservar memoria para los datos enviados por la cámara, configurando parámetros como los tipos de buffer que se usarán y su tamaño. Una vez configurado se hace una petición `VIDEOC_STREAMON`.

Una vez seguidos los pasos anteriores ya se pueden obtener fotogramas de la webcam, para lo que se implementa una función para obtenerlos. En ella, se bloquea el programa hasta que hay datos listos y entonces se hace una petición `VIDEOC_DQBUF` para obtenerlos.

Para cerrar la conexión simplemente hace falta enviar una petición `VIDEOC_STREAMOFF` y liberar todos los recursos reservados.

En OpenCV usando el objeto de la clase `VideoCapture` se puede acceder a fotogramas. Para ello, el objeto tiene tres funciones clave: `open`, `read` y `close` que respectivamente sirven para abrir la conexión, leer fotogramas y cerrar la conexión. Lo único importante a tener en cuenta es que para leer fotogramas hace falta utilizar la clase `Mat` de OpenCV como contenedor.

Para escribir los fotogramas en disco se utiliza OpenCV exclusivamente, en este caso se utiliza la clase `VideoWriter`. Se inicializa usando su constructor y parámetros varios como el tamaño de los fotogramas y el códec de vídeo.

Para escribir los fotogramas es necesario usar la ya mencionada clase `Mat`. Para poder utilizarla con fotogramas obtenidos de V4L2 simplemente hace falta crear un objeto nuevo de esta clase y cambiar su puntero de datos por un puntero a los datos del fotograma obtenido. Para escribir los fotogramas al fichero representado por el objeto simplemente hay que invocar la función `write` con la estructura que contiene el fotograma. Por último, una vez finalizado el vídeo es necesario cerrar el fichero usando la función `release` porque en caso contrario el fichero se corromperá y no se podrá visualizar correctamente.

Este proceso de adaptación se puede observar en el siguiente fragmento de código de ejemplo:

```
1. int ancho, alto;
2. char *fotograma; //almacenamiento del fotograma
3. ...
4. cv::VideoWriter video("salida.avi", cv::VideoWriter::fourcc('M', 'J', 'P', 'G'), 30, cv::Si
   ze(ancho, alto));
5. ...
6. cv::Mat fotogramaCV(ancho, alto, CV_8UC3, fotograma); //Creacion estructura
7. fotogramaCV.data = fotograma;
8. video.write(fotogramaCV);
9. video.release();
```

Por último, para visualizar imágenes en OpenCV es necesario tener instalado y funcionando un entorno de ventanas. Para mostrar los fotogramas basta con utilizar la función `imshow` con una estructura Mat como parámetro, seguidamente hace falta llamar a `waitKey` para que muestre la imagen.

3.3 Procesamiento de vídeo

Para construir el programa principal, el siguiente paso es utilizar todos los módulos generados para obtener fotogramas, procesarlos y escribirlos en disco utilizando diferentes estrategias de paralelismo.

El software recibe como parámetros de entrada lo siguiente:

- Dispositivo o fichero para leer.
- Alto y ancho del fotograma.
- Duración de la grabación y tasa de fotogramas por segundo.
- Filtro que se va a aplicar, puede ser gauss, escala de grises o ninguno (simplemente graba).

En el programa se usan las librerías de sistema para obtener marcas de tiempo y así medir el rendimiento de las regiones de interés del código. Estos datos se muestran por la salida estándar en formato CSV.

En tiempo de compilación se puede elegir la estrategia de paralelización: paralelismo a nivel de datos, a nivel de tareas o explotado por OpenCL. Por defecto se compila la versión de OpenCL.

3.4 Mejorando el rendimiento

Una vez realizado el programa y comprobado que funciona correctamente, el siguiente objetivo principal es mejorar su rendimiento todo lo posible teniendo una versión secuencial sin optimizaciones específicas. Lo primero que se puede hacer es en tiempo de compilación, utilizar la directiva "O3" del *GNU Compiler Collection* (GCC), la cual aplica una serie optimizaciones de forma automática, principalmente la vectorización de bucles.

Lo siguiente es paralelizar el código puesto que es una forma relativamente sencilla de obtener mejor rendimiento. A continuación, se presentan tres posibles paralelizaciones: dos sobre la CPU utilizando OpenMP como modelo de programación y otra usando la OpenCL como modelo de programación.

3.4.1 Paralelismo de datos

La primera optimización es aplicar OpenMP para explotar el paralelismo de datos en los filtros. Para ello se usará la cláusula `for` que se ha explicado en la sección 2.1.3 dentro de los filtros, estableciendo como sección paralela la totalidad del filtro. Entonces, cada filtro tiene una sección paralela en la que se divide el trabajo según el planificador utilizado. En este proyecto se utiliza el planificador dinámico `schedule(dynamic)` con el fin de aprovechar al máximo la CPU puesto que para optimizar las escrituras se usarán también hilos. Al tener 5 hilos habrá ciertas interferencias por el planificador de hilos y esto aumenta la latencia si se utiliza planificador estático en OpenMP. Utilizando el planificador dinámico se reduce el impacto del quinto hilo.

Para optimizar las escrituras como ya se ha mencionado, se utiliza un hilo de la librería estándar de C++ que escribe los fotogramas procesados a disco con el objetivo de no bloquear la lectura y procesamiento del siguiente fotograma. Para ello es necesario reservar memoria para dos buffers donde almacenar los fotogramas leídos y los procesados. Es necesario que se utilice este hilo en vez de OpenMP porque debe estar situado en el programa principal. Si declarase una sección paralela dentro del programa principal, la declarada en el filtro produciría un comportamiento indefinido.

Este programa es un ejemplo claro del problema productor-consumidor, el programa principal (productor) genera los fotogramas procesados y avisa al hilo (consumidor) que hay fotogramas listos para escribir en disco. En caso de no tener espacio en sus buffers para más fotogramas, este se bloquea hasta que reciba una señal por parte del hilo. Si ya terminó la grabación y procesamiento, el programa principal envía una señal de finalización al hilo.

El hilo simplemente inicializa un fichero para guardar el video en disco, seguidamente se dedica a escribir los fotogramas en disco hasta que el programa principal le envíe la señal para parar. Si el hilo detecta que los buffers han pasado de estar llenos a tener hueco, este avisa con una señal al programa principal. Este hilo también puede usarse para que muestre los fotogramas en pantalla pudiendo también superponer cosas como la tasa de fotogramas por segundo o la versión que se está ejecutando.

Es altamente importante que este hilo se inicialice fuera de secciones `parallel` de OpenMP puesto que de otra forma se pueden generar comportamientos indefinidos.

3.4.2 Paralelismo de tarea

Una vertiente diferente a la anterior es utilizar la cláusula `task` también explicada en la sección 2.1.3 aunque esta vez se aplica en el programa principal. La sección paralela es el lazo en el que se extraen, procesan y almacenan los fotogramas. Se generan dos tipos de tareas, de ejecución de filtro y guardado de fotogramas en disco.

Lo que se consigue en este programa explotando el paralelismo a nivel de tareas es un flujo de trabajo en el cual se leen fotogramas y se procesan de forma simultánea. Lo cual lleva a una situación en la que varios hilos están procesando fotogramas de forma simultánea. Además, esta versión puede escribir en disco mientras se procesan y leen fotogramas sin tener que recurrir a usar hilos de la librería estándar.

Esta versión también necesita buffers para almacenar fotogramas de forma temporal con el objetivo de no perder información, en esta versión se utilizan cuatro en vez de dos, dos para fotogramas sin procesar y dos para filtrados. Mientras se leen y procesan imágenes se utiliza solo uno de cada tipo y cuando estos están llenos se genera una tarea de escritura en disco y se comienza a usar los otros buffers libres.

En las siguientes ilustraciones se mostrará la diferencia entre utilizar paralelismo de datos y paralelismo de tareas.

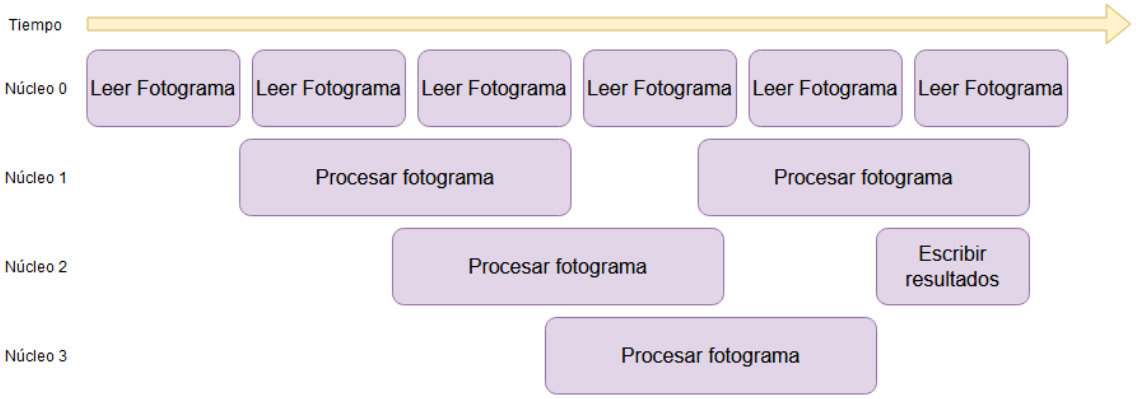

Figura 3.3: Diagrama de ejecución en la versión de paralelismo de tareas

A medida que avanza el tiempo, el núcleo 0 de la CPU lee fotogramas de la cámara de vídeo y genera tareas. Cuando se generan tareas. Si hay un núcleo libre, este comenzará a procesar el fotograma aplicando el filtro seleccionado. En este ejemplo se usan buffers de 3 fotogramas de tamaño, por eso mismo cuando termina el tercer núcleo de procesar, en el segundo comienza una tarea de escritura de resultados.

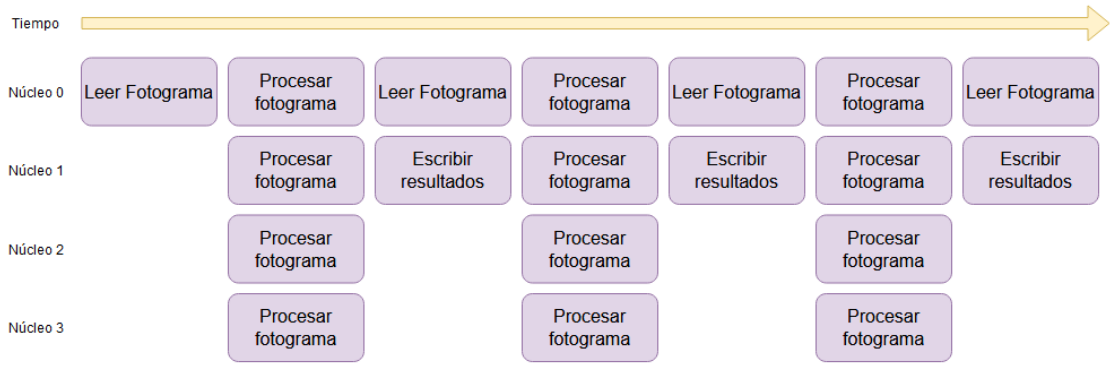

Figura 3.4: Diagrama de ejecución en la versión de paralelismo de datos

En este caso, en paralelismo de datos, el núcleo 0 lee un fotograma y este se procesa en los 4 núcleos de forma simultánea, la siguiente iteración se lee el fotograma siguiente y se almacena el resultado anterior a la vez. Este proceso sigue hasta que se ha grabado y procesado el vídeo completo.

3.4.3 Paralelismo explotado por OpenCL

La siguiente versión es la que explota el paralelismo mediante el uso de OpenCL y el módulo generado en la sección 3.1.3. Esto permite ejecutar los filtros programados tanto en CPU como en GPU.

Con el objetivo de no tener un programa principal complejo y poco mantenible se optó por utilizar el módulo intermedio generado en la sección 3.1.

Para cada filtro existen dos funciones: la primera función prepara la ejecución del *kernel* configurando argumentos, tamaños y otros parámetros necesarios. La segunda función envía los datos de entrada, encola la orden de ejecutar el *kernel,* esperando a que este termine su ejecución y, por último, recibe los datos de salida.

Esto todavía puede ser optimizado y para ello primero se utilizarán los datos de tipo evento proporcionados por OpenCL.

Muchas de las funciones de OpenCL retornan un dato del tipo `cl_event`, este dato se utiliza para no bloquear el programa de forma innecesaria. Una forma de aplicar esto para obtener mejor rendimiento es que las escrituras y lecturas de datos al dispositivo no sean bloqueantes. Los eventos retornados se pueden usar en la función `clEnqueueNDRangeKernel` como parámetro y provoca que la ejecución del kernel espere a que estos eventos finalicen. Teniendo entre el envío de datos y la llamada a `clEnqueueNDRangeKernel` la capacidad de hacer otras operaciones.

Otra optimización es que la propia ejecución del kernel no sea bloqueante y que el flujo de trabajo sea el siguiente:

1. Leer un fotograma.
2. Obtener resultados del fotograma procesado anterior (en caso de existir).
3. Enviar datos al dispositivo y ejecutar kernel no bloqueante.
4. Recuperar fotograma procesado y almacenar en disco.
5. Repetir 1-4 hasta finalizar la grabación.

Para ello la función del módulo intermedio que ejecuta el kernel se divide en dos partes, ejecución del kernel y esperar a que finalice. Consecuentemente, se utiliza el evento retornado por `clEnqueueNDRangeKernel` para evitar el bloqueo.

Por último, esta versión también puede mejorar reutilizando el hilo de la librería estándar de C++ que se creó en la sección 3.4.1. En este caso no podemos usar paralelismo de tareas de OpenMP para esto porque si el *kernel* lanzado por OpenCL se ejecuta en CPU puede generar sobresuscripción, que haya más hilos creados de los que puede ejecutar el sistema de forma simultánea. Esto puede provocar perdida de rendimiento. El uso de este hilo es el mismo, de forma paralela se almacenan los fotogramas en disco.

Como última prueba de rendimiento para comprobar si la plataforma elegida es capaz de computar procesamiento de vídeo en tiempo real fue portar un kernel que realiza una transformación proyectiva.

Para implementarlo se hallaron varios problemas:

- La GPU Videocore IV de Raspberry Pi 3 no puede ejecutar operaciones de 64 bits. Siendo problemático porque el kernel original utilizaba números en punto flotante de doble precisión. Para arreglarlo fue necesario cambiar el tipo de dato a un número flotante de simple precisión que ocupa 32 bits. Además de modificar el *kernel* también fue necesario modificar el paso de parámetros, en particular la lista de números decimales.
- La implementación original utilizaba tipos de datos que el compilador no reconocía y fue necesario adaptarlos creando estructuras y aliases. Realizando un seguimiento del código de *kernel* y su lanzamiento en su implementación original se pudieron inferir todos los tipos que faltaban.

- La implementación de OpenCL para Videocore IV no soporta *kernels* bidimensionales de forma correcta. Lo que sucedía es que el controlador de OpenCL para GPU no generaba suficientes *work-items*, mientras que en CPU funcionaba sin problemas. Puesto que el kernel no usaba internamente los índices de *work-group*, se pudieron generar los índices de fila y columna usando un solo índice en un *kernel* unidimensional. Para solucionarlo, se divide el índice global entre el número de filas de la imagen, el cociente representa al índice de fila y el resto al índice de columna.

Por lo tanto, dos de tres problemas son debidos a la Raspberry Pi y su entorno, esto se ha expuesto subrayando la dificultad de obtener el máximo rendimiento en plataformas embebidas. Es cierto que el soporte de CPU y Linux es excelente, pero utilizar la GPU es problemático y no parece que el sistema tuviera en cuenta la posibilidad de explotar este dispositivo para cómputo.

Lo que se ha expuesto este proyecto representa una parte de la funcionalidad de OpenCL. La alta complejidad de implementarlo en una plataforma como Raspberry Pi es lo que provoca los problemas encontrados. Y por este mismo motivo, la implementación de OpenCL en Videocore IV no funciona correctamente en los casos estudiados.

Capítulo 4: Experimentación

4.1 Metodología

Como se ha mencionado previamente, la experimentación se ha ejecutado en una Raspberry Pi modelo 3B con un procesador de 4 núcleos que trabajan a 1.2GHz, una GPU VideoCore IV y 1 GB de memoria RAM.

El nodo X86 utiliza un procesador AMD A10 7850k Radeon R7, 3.7GHz con dos núcleos físicos y capacidad de ejecutar 2 hilos por núcleo, gráficos integrados y 8 GB de memoria RAM.

En el nodo se ha utilizado una Webcam Logitech C290 mientras que en la Raspberry Pi se ha utilizado una Webcam Netway NW798.

En los experimentos que no se especifique, los vídeos tienen un tamaño de fotograma de 640x480 con una longitud de 200 fotogramas. El tamaño de los buffers de fotogramas utilizado es de 10 fotogramas.

La principal métrica son los fotogramas por segundo, ya sea utilizando como entrada cámaras USB, con sus límites de 30 FPS, o ficheros de vídeo, que no están limitados.

En ambas máquinas todo lo innecesario fue desactivado. como la interfaz gráfica puesto que pueden interferir con las medidas, aunque para los experimentos que la necesiten sí fue activada.

Para cada experimento se ha generado un script que automáticamente ejecuta la aplicación con los parámetros considerados adecuados además de recoger sus resultados en un fichero de texto.

Cada escenario se ha ejecutado doce veces de las cuales se descartan los datos de la primera y la última ejecución.

En la mayoría de los casos se ha utilizado un tamaño de vídeo de 640x480 píxeles con 200 fotogramas de duración y se han ejecutado todos los filtros relevantes para el experimento.

4.2 Experimentos desarrollados

4.2.1 Línea base de obtención de fotogramas

En este experimento se midió el rendimiento de las demos generadas en la sección 3.2 configuradas solo para leer fotogramas, sin escribirlos en disco ni mostrarlos por pantalla, con el objetivo de observar la máxima tasa de fotogramas por segundo que podría alcanzarse en este sistema. Esto se comprobó en la Raspberry Pi puesto que es el sistema principal para el que se ha optimizado el software.

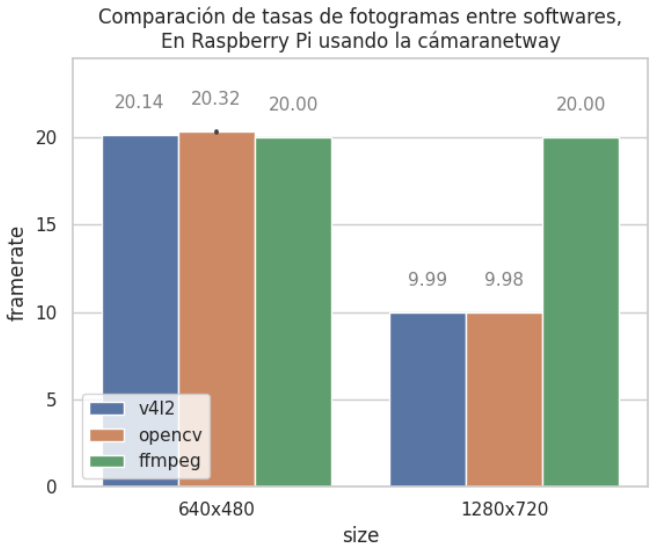

Figura 4.1: Comparación entre librerías y programas de obtención de vídeo

Como se puede observar en la *Figura 4.1*, la demo que utiliza Video 4 Linux y la demo que utiliza OpenCV obtienen el mismo rendimiento que el software FFmpeg en vídeos con tamaño de fotograma de 640x480. Las tres aplicaciones obtienen el mismo rendimiento de 20 fotogramas por segundo lo cual es inconveniente puesto que la cámara utilizada debería alcanzar su máximo de 30 fotogramas por segundo (FPS). En computadores de sobremesa esta cámara sí alcanza el máximo de 30 FPS conectándola a través de puertos USB 2.0.

En grabaciones con tamaño de fotograma 1280x720 ambas librerías obtienen 10 FPS mientras que FFmpeg obtiene 20 FPS. Si comparamos el rendimiento entre demos se puede suponer la demo de V4L2 no está mal optimizada puesto que obtiene el mismo resultado que la demo de OpenCV, que al ser de código abierto y ampliamente usada debería estar muy optimizada.

Comparando los resultados de V4L2 y OpenCV en este caso se puede suponer que el módulo programado de V4L2 no debería está mal optimizado puesto que obtiene el mismo rendimiento que una librería popular de código abierto.

La diferencia con FFmpeg en este caso se debe a que la grabación de FFmpeg utiliza el formato comprimido MJPEG y no se descomprime ni decodifica. Mientras que las demos obtienen los fotogramas y los decodifican a un formato sin compresión RGB. En esta aplicación usar MJPEG no era posible sin tener que modificar completamente el comportamiento de los filtros, las reservas de memoria y la obtención del vídeo. Este cambio requería mucho trabajo además de aumentar la complejidad de trabajar con datos comprimidos.

Es posible que para *kernels* o filtros sencillos MJPEG sea interesante puesto que los datos ocupan menos espacio y por lo tanto tiene menor latencia al enviarlos a un dispositivo.

En el caso de usar un tamaño de 640x480 la imagen era lo suficientemente pequeña así que la decodificación no suponía un impacto grande en el rendimiento, pero aumentar el tamaño del fotograma supone perder rendimiento.

Respecto al límite de 20FPS puede ser por varios motivos, principalmente dos:

- Falta de ancho de banda en los puertos USB, la Raspberry Pi tiene 4 puertos USB 2.0 multiplexados con el bus ethernet.
- Falta de potencia, puede que la alimentación proporcionada por los USB de la Raspberry no sea suficiente para la Webcam.

En ningún caso la CPU supera el 30% de uso por lo que no parece que la falta de cómputo en la CPU sea el problema.

4.2.2 Comparación entre las primeras versiones

Este experimento se ha realizado con el objetivo de justificar la necesidad de optimizar el programa en la medida de lo posible.

La prueba consiste en ejecutar la aplicación siguiendo tres modelos de cómputo distinto. Se ejecuta en la Raspberry Pi usando los filtros implementados y comparar sus rendimientos usando los mismos parámetros de entrada.

En la *Figura 4.2* podemos ver los resultados de la versión secuencial (SEQU), basada en paralelismo de tareas (OMPT) y basada en paralelismo de datos (OMPP).

Como se puede ver, el filtro de desenfoque gaussiano obtiene una mejora de rendimiento grande (2.8 veces más rápido) mientras que el filtro de escala de grises obtiene una mejora pequeña (1.2 veces más rápido).

Estas diferencias en la obtención de la mejora del rendimiento se pueden explicar con la ley de Amdahl: la mejora de rendimiento obtenida en un programa por optimizar un componente o una parte de un programa está limitada por la fracción de tiempo que toma ejecutar esa parte. En este caso, el desenfoque gaussiano ocupa una fracción del tiempo total de ejecución mayor que el filtro de escala de grises por lo que la mejora de rendimiento es mayor.

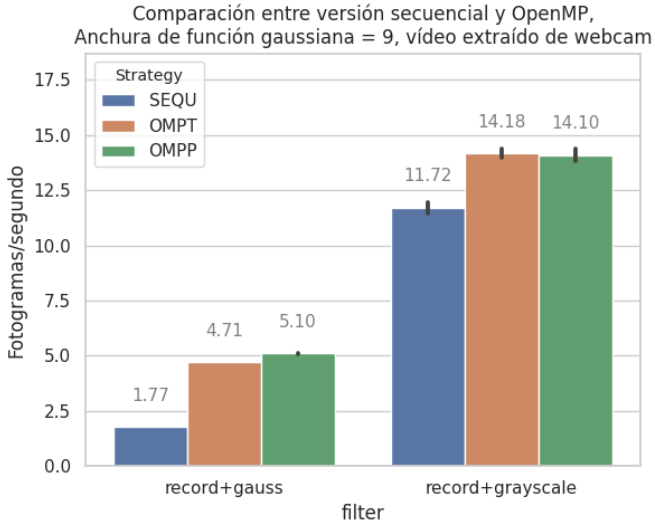

Figura 4.2: Comparación entre diferentes versiones del programa

Si se disminuye el ancho de la función gaussiana se puede lograr una tasa de fotogramas más alta, aunque la mejora frente a la versión secuencial será más baja. Además, reducir este ancho impacta a la efectividad del desenfoque generando fotogramas con menor desenfoque.

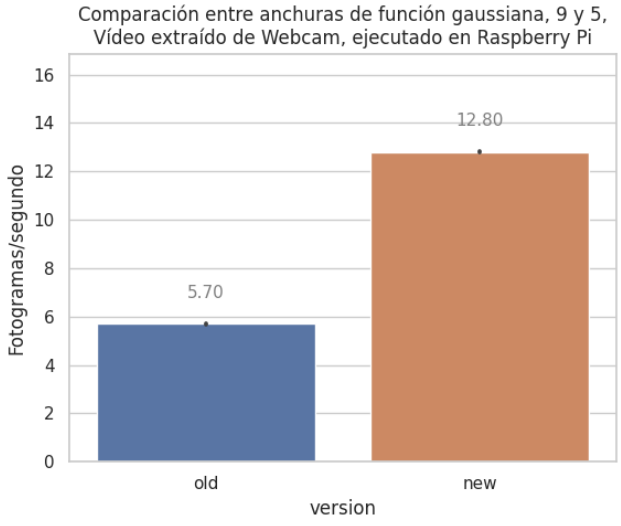

Figura 4.3: Comparación entre rendimiento de diferentes anchos de función gaussiana

Como podemos ver en la *Figura 4.3*, tener un filtro menos efectivo es una gran mejora de rendimiento, aunque como hemos visto esto afecta a la efectividad del paralelismo.

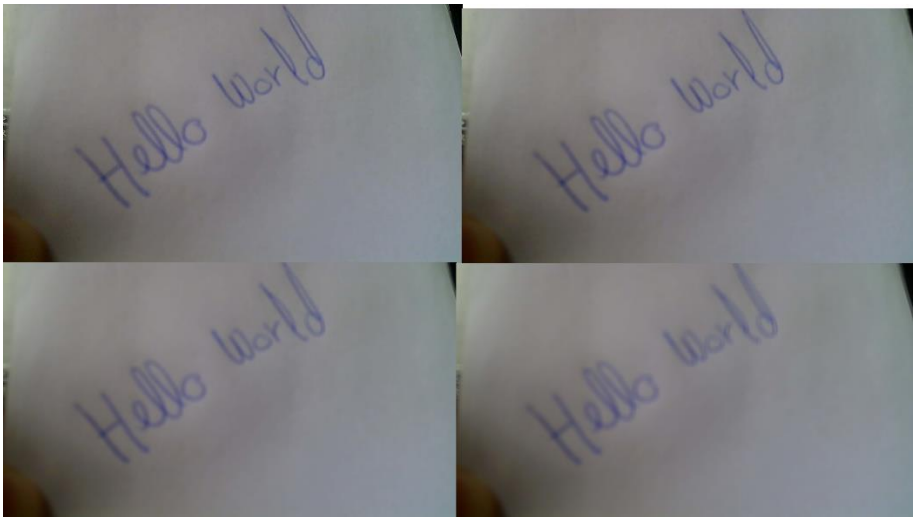

Figura 4.4: Comparación entre diferentes anchos de función gaussiana

En la *Figura 4.4* podemos ver la efectividad del filtro, la primera imagen es la entrada sin procesar y las siguientes tres imágenes son las procesadas con funciones crecientemente efectivas. Las imágenes en la esquina superior derecha y la esquina inferior izquierda corresponden a las realizadas en el experimento.

4.2.6 Usar instrucciones vectoriales explotando el paralelismo de datos.

Una optimización que se puede aplicar a la versión basada en paralelismo de datos es utilizar instrucciones vectoriales de forma que varias iteraciones de un bucle se ejecuten de forma simultánea en un solo hilo.

Esto se consigue con la cláusula “SIMD”. Se puede ver en ejemplo de uso en el siguiente código [29]:

```
1. #pragma omp simd
2. for (auto i = 0; i < count; i++)
3. {
4.    a[i] = a[i - 1] + 1;
5.    b[i] = *c + 1;
6. }
```

En principio si la directiva “O3” del compilador funciona bien esto no debería proporcionar una mejora de rendimiento, pero es posible que GCC no aplique esta vectorización porque no sea capaz de detectar la posibilidad.

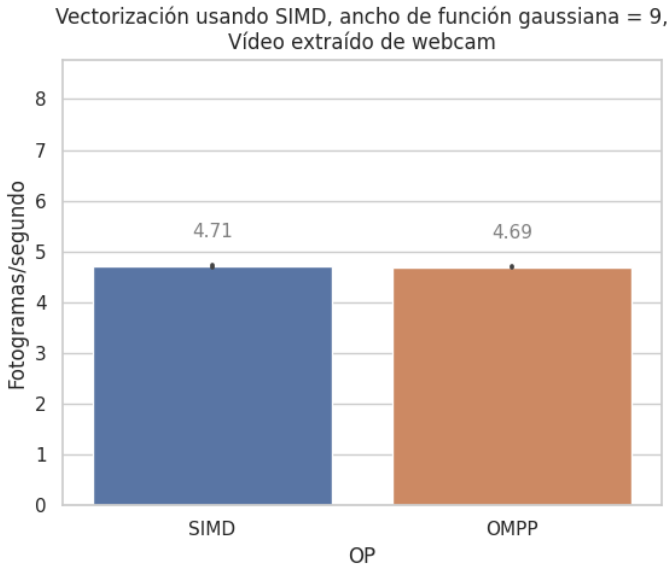

Figura 4.5: Comparación entre utilizar directiva SIMD o no.

En la *Figura 4.5* en la columna “SIMD” vemos los resultados de aplicar la optimización y la columna “OMPP” vemos los resultados originales. Como se puede observar esta optimización no era necesaria porque el compilador ya la estaba aplicando sin necesidad de recurrir a OpenMP.

Considerando que el desenfoque gaussiano es más complejo que la conversión a escala de grises se puede asumir que el segundo filtro también se vectoriza de forma correcta.

4.2.3 Tamaño óptimo del buffer en la versión de tareas

Para la versión basada en el paralelismo de tareas usar un tamaño de buffer demasiado grande o pequeño afecta al rendimiento puesto que puede generar bloqueos de dos formas:

1. Si el buffer es muy grande. La tarea de escritura es muy larga y el programa tiene que esperar a que termine antes de lanzar la siguiente escritura.
2. Si el buffer es muy pequeño entonces puede que el programa se bloquee porque los buffers se llenan muy rápido y hace falta lanzar tareas de escritura constantemente.

Así, el siguiente experimento trata de determinar este tamaño óptimo.

Como se puede ver en la *Figura 4.6* los tamaños de buffer de 5, 10 y 20 fotogramas son los que mejor funcionan, aunque las diferencias con buffers más grande son mínimas.

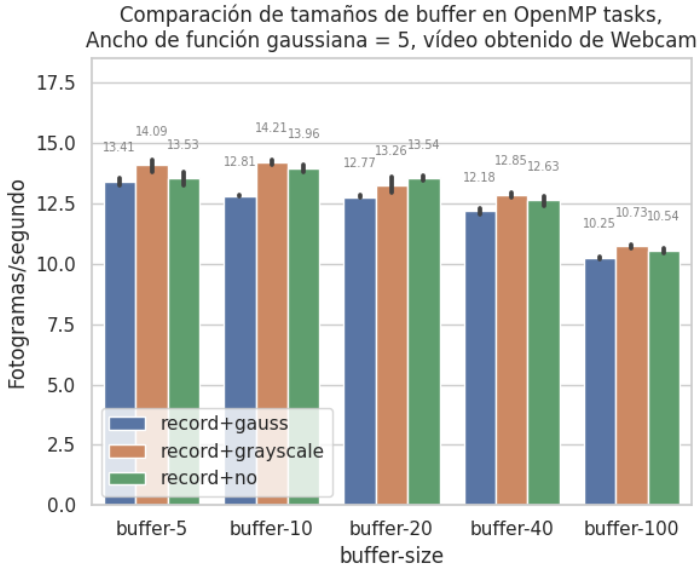

Figura 4.6: Comparación de tamaño de buffer en versión de paralelismo de tareas

4.2.4 Paralelismo utilizando GPU integrada

El objetivo de este experimento es comprobar si se puede optimizar más el código para ejecutarse en la Raspberry Pi 3 utilizando la GPU integrada del sistema. La siguiente prueba comprueba la diferencia entre ejecutar en CPU o GPU, ambas usando OpenCL.

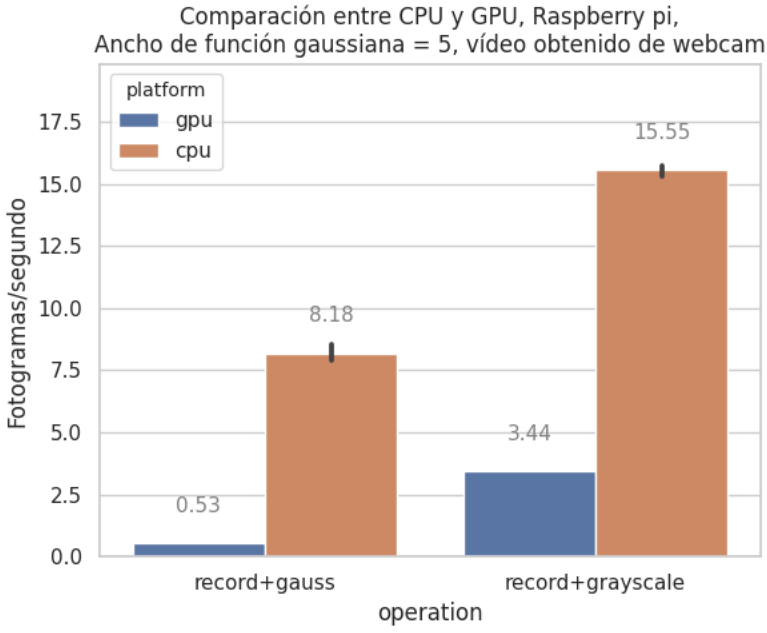

Figura 4.7: Comparación entre GPU y CPU en Raspberry Pi

Como se puede observar en la *Figura 4.7* no parece que la GPU integrada de esta plataforma sea buena para cómputo.

Es posible que ocurra por uno de varios motivos, como la latencia entre el paso de datos de la memoria principal a la memoria de la GPU, porque la unidad de cómputo es lenta o porque la implementación de OpenCL en el controlador de la GPU no es buena.

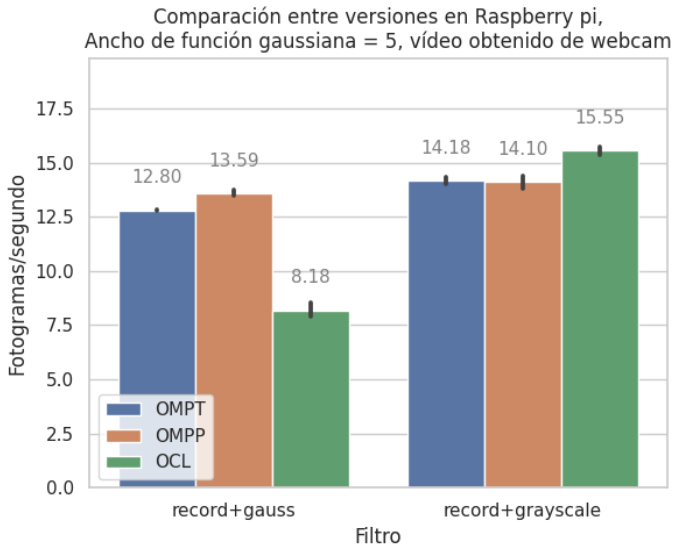

Figura 4.8: Comparación entre versiones

En la *Figura 4.8* se muestra que OpenCL (denotado OCL) obtiene peor rendimiento en el desenfoque gaussiano, pero en el filtro de escala de grises obtiene mejor. Esto puede deberse a que la complejidad del desenfoque cause que OpenCL no paralelice de forma óptima este filtro.

4.2.5 Longitud del vídeo

El objetivo de este experimento es comprobar si la longitud del vídeo tiene impacto sobre el rendimiento utilizando OpenCL. Esto se hace con el objetivo de comprobar si existe algún caso en el que la GPU pueda ser mejor opción que la CPU.

Se puede ver en la *Figura 4.9* los resultados obtenidos por plataforma y longitud de tiempo, cada barra representa una plataforma y una longitud de vídeo. Por ejemplo, “cpu5s” significa que se ha ejecutado sobre CPU y se ha grabado un vídeo de 5 segundos.

Salvo en el caso de ejecutar desenfoque gaussiano en CPU, no parece que haya ninguna diferencia al ejecutar el programa con diferentes longitudes de vídeo.

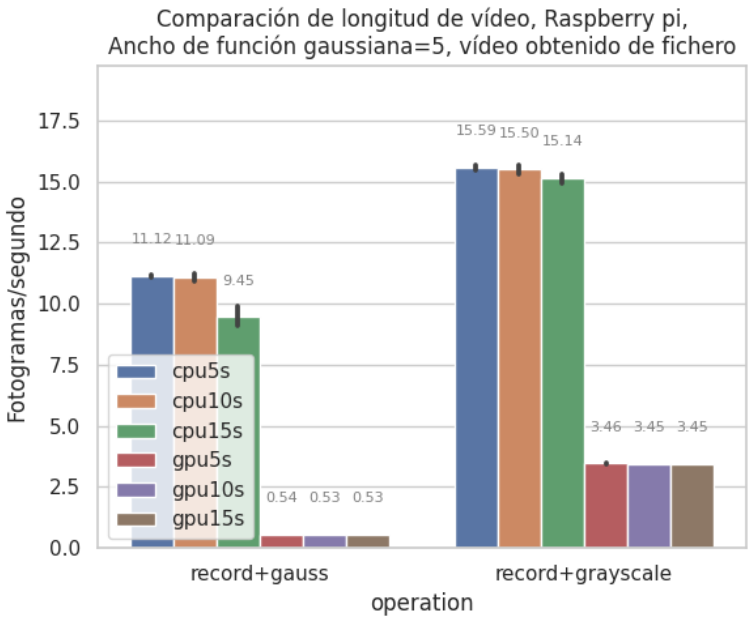

Figura 4.9: Comparación entre diferentes longitudes de vídeo por dispositivo

4.2.7 Evitar bloqueos de escritura y de ejecución en OpenCL

Una de las primeras optimizaciones que se pueden aplicar a OpenCL es evitar los bloqueos en los envíos de datos. Se pueden encolar las escrituras de datos de entrada de forma no bloqueante y simultáneamente extraer el siguiente fotograma de la cámara mientras se ejecuta el *kernel*.

Para lograrlo se utilizaron los eventos proporcionados por OpenCL como se detalló en la sección 3.4.3.

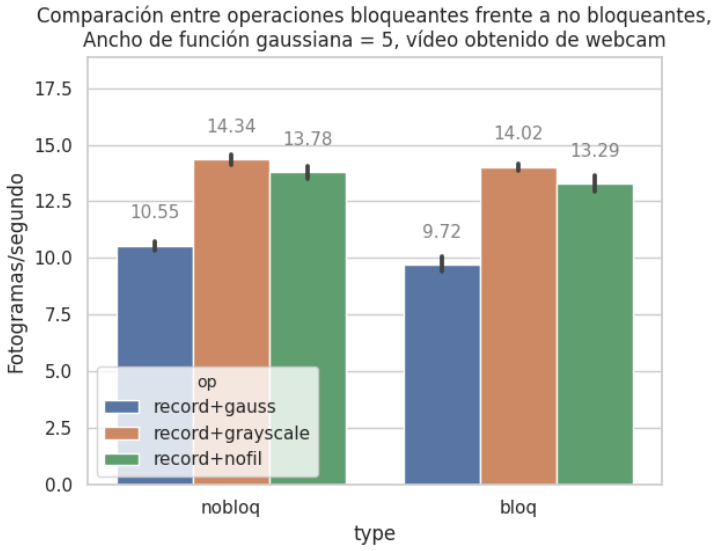

Figura 4.10: Comparación entre operaciones bloqueantes o no bloqueantes de OpenCL

Se puede observar en la *Figura 4.10* que estas optimizaciones no han tenido el impacto esperado. Esto se debe por dos cosas, entre el envío de datos y la ejecución del kernel no hay ninguna operación que se pueda computar para aprovechar este tiempo y segundo, una vez ejecutado el kernel, traer los datos de vuelta antes de procesar el siguiente fotograma supone perder el tiempo ganado al ejecutar el kernel mientras se obtiene el siguiente fotograma.

4.2.8 Optimalidad del software generado

Para comprobar y justificar que la aplicación generada no está mal optimizada o desarrollada se ha realizado este experimento en el que las tres versiones (OpenCL, y ambas vertientes de OpenMP) se comparan entre la Raspberry Pi y el otro computador x86 descrito en la introducción de este capítulo.

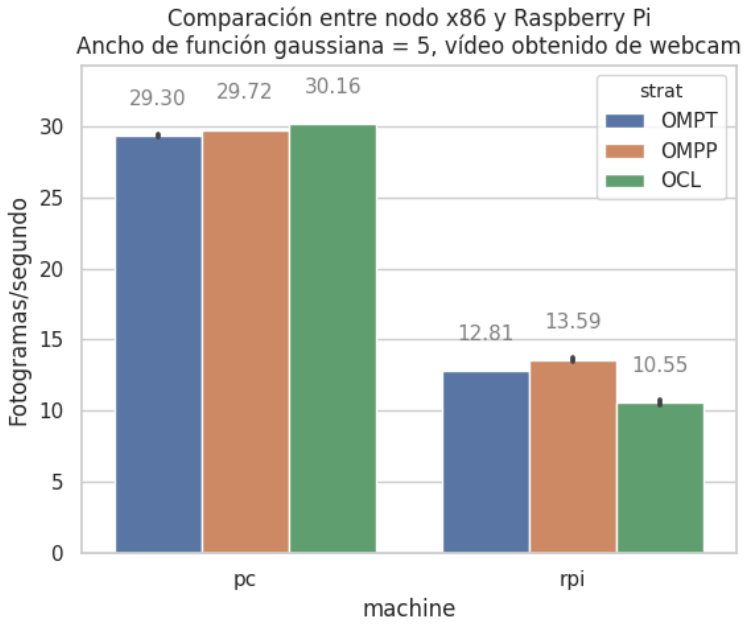

Figura 4.11: Comparación entre Raspberry Pi y nodo x86

Como se puede observar en la *Figura 4.11*, el software desarrollado alcanza el límite de 30 fotogramas por segundo de la cámara en el computador x86. Esto deja claro que la Raspberry no tiene una capacidad de cómputo suficiente para permitir el procesado de vídeo en tiempo real. Es interesante destacar que en este la versión con mejor rendimiento es la realizada en OpenCL, frente a las ejecutadas en la CPU.

4.2.9 Máxima capacidad del software

Este experimento tiene como objetivo comprobar cuanto rendimiento puede extraer la versión de OpenCL utilizando un fichero de vídeo en vez de una cámara. La prueba se ha ejecutado sobre el computador x86 utilizando sus gráficos integrados.

Además, las medidas realizadas han consistido en utilizar diferentes tamaños de *work-group* explicados en la sección 3.1.3 (denominado "block-size" en la figura) con el objetivo de obtener el máximo rendimiento posible.

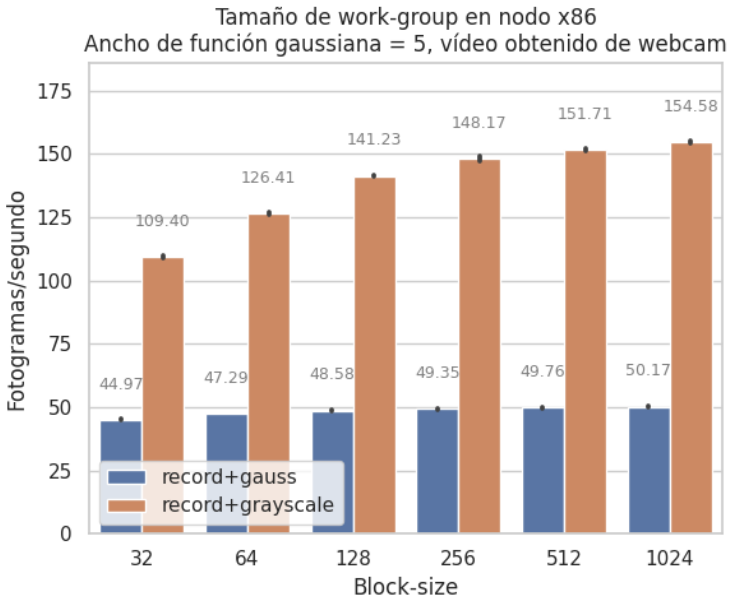

Figura 4.12: Comparación entre múltiples tamaños de work-group

Como se puede observar, el programa todavía tiene capacidad para procesar fotogramas con filtros o algoritmos que requieran más cómputo sin obtener una tasa de fotogramas menor que 30 FPS.

También se puede observar que los tamaños de bloque más grandes (256, 512 y 1024) son los mejores en este caso. Esto ocurre por dos motivos, primero, los bloques grandes compensan mejor la sobrecarga en las comunicaciones. Además, hacen mejor uso del hardware subyacente aprovechando al máximo todas las unidades de cómputo disponibles.

4.2.10 Aplicar MapBuffer

Este experimento tiene como objetivo mejorar el rendimiento de la versión basada en OpenCL reduciendo el impacto del intercambio de datos. El experimento se ejecutó en el computador x86 utilizando un vídeo de 1280x720.

Para ello se utiliza la función `clEnqueueMapBuffer`, con ella se asigna una región de memoria del dispositivo en el cliente (la memoria principal). Entonces los datos se escriben directamente en esa zona asignada usando la función `memcpy`. Una vez escritos se usa la función `clEnqueueUnmapObject` para liberar la asignación. Con ello el dispositivo puede acceder a los datos sin necesidad de tener que haberlos enviado desde la memoria principal usando `clEnqueueWriteBuffer`.

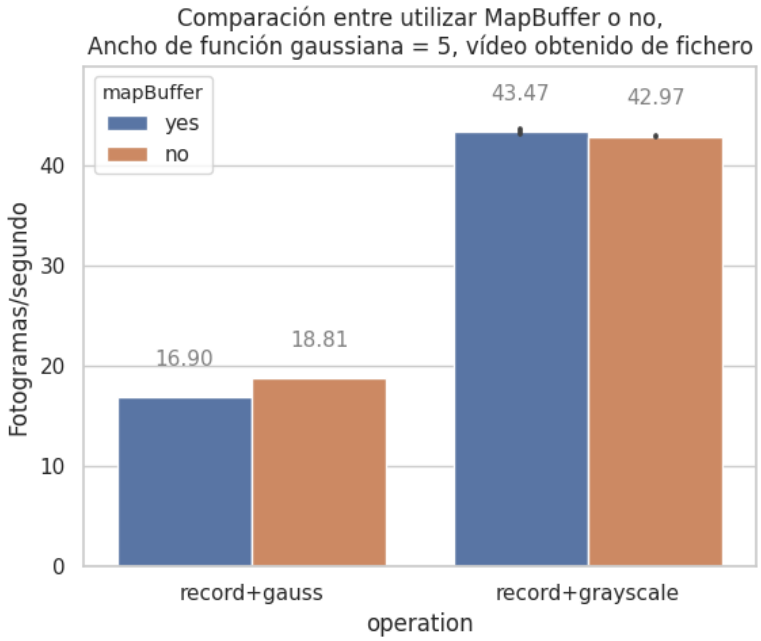

Figura 4.13: Comparación entre usar o no MapBuffer en OpenCL

En la *Figura 4.13* vemos que el resultado apenas tiene impacto. Esto significa que la latencia del envío de datos ya era los suficientemente pequeña puesto que estamos trabajando con GPU integrada y tampoco hay posibilidad de utilizar *zero-copy buffers.*

4.2.11 Warp Perspective

Este último experimento tiene como objetivo exponer la capacidad de cómputo de la Raspberry Pi y la potencia de OpenCL. Para ello se utiliza un algoritmo de procesamiento de vídeo que realiza una transformación proyectiva.

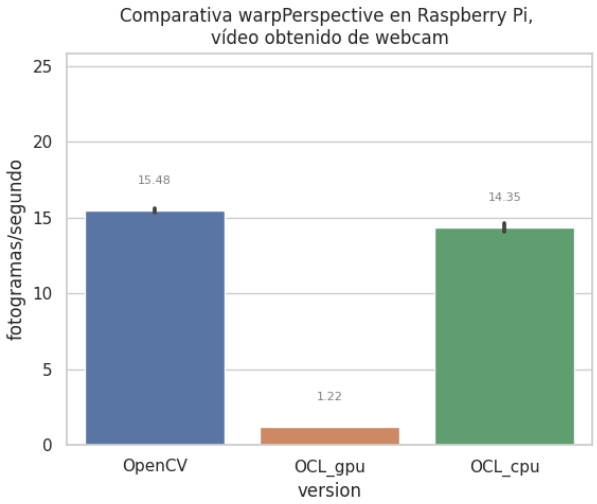

Figura 4.14: Comparativa entre OpenCL (GPU y CPU) frente a Opencv

En la anterior figura se puede observar cómo OpenCL ejecutado en CPU apenas tiene diferencia de rendimiento. El rendimiento de ambas versiones proviene del mismo lugar, explotar el paralelismo utilizando múltiples hilos en CPU.

Esta igualdad de rendimiento es muy beneficiosa porque supone tener portabilidad a cualquier sistema heterogéneo que soporte las mismas características, pero, sin sacrificar el rendimiento.

En el caso de la GPU se vuelve a observar que Videocore IV no es apropiada para cuestiones de cómputo.

Capítulo 5: Conclusiones

El objetivo principal de este proyecto era comprobar la capacidad que tiene la plataforma Raspberry Pi para computar video en tiempo real y si se podía mejorar el rendimiento utilizando su GPU.

Paralelizar los filtros de fotogramas usando OpenMP es una tarea relativamente rápida, aunque no trivial, debido a la facilidad de uso que proporciona este estándar. Además, su sencillez no impide obtener notables mejoras de rendimiento. Para sistemas homogéneos que no utilicen aceleradores OpenMP es una opción muy competente además de ser portable.

Por otro lado, paralelizar utilizando OpenCL es un proceso que lleva mucho tiempo. Primero, el proceso de configuración y ejecución requiere muchas líneas de código y comprobación de errores, derivando en problemas de mantenibilidad. También requiere comprender los aspectos técnicos que conlleva la ejecución de un *kernel* para asegurar que funciona correctamente y con un rendimiento óptimo. Otro problema que se puede encontrar al utilizar OpenCL es la instalación del entorno y drivers. Esta misma fue proporcionada al comienzo del trabajo debido a la alta dificultad del proceso y el tiempo que requiere invertir. También se debe mencionar que OpenCL no proporciona ninguna herramienta para depurar *kernels*, lo cual entorpece el desarrollo de código paralelo usando este estándar. OpenCL es una herramienta muy potente pero excesivamente compleja debido a su compatibilidad con todo tipo de sistema heterogéneo.

Respecto al entorno de la Raspberry Pi, todas las herramientas usadas en este proyecto han funcionado sin problema alguno. Incluso alguna herramienta contemplada para usar en este proyecto como puede ser *perf* en Linux es compatible. Entonces en términos de depuración y evaluación de rendimiento, la experiencia no es distinta a otros sistemas Linux o Unix.

Como OpenCV no proporcionaba el rendimiento adecuado, se optó por usar V4L2. Además de ser bastante más complejo que OpenCV, obtiene el mismo rendimiento. Por lo tanto, a menos que haya algún problema o requerimiento que no permita utilizar librerías como OpenCV, utilizar directamente Video 4 Linux no es recomendable.

Respecto a la GPU de la Raspberry Pi, en el estado actual de la plataforma tanto por la implementación de OpenCL en el controlador o limitaciones de la propia GPU, ejecutar código sobre ella no resulta especialmente interesante, aunque es cierto que con balanceo de carga se puede obtener más rendimiento en el sistema. Los tres factores más limitantes son la incapacidad de ejecutar *kernels* multidimensionales correctamente, no poder utilizar operaciones de 64 bits y el bajo rendimiento obtenido.

Por otro lado, es notable la capacidad de cómputo que ofrece la Raspberry Pi usando hilos de CPU, durante el desarrollo de los programas OpenMP se ejecutaron los mismos programas en un portátil con procesador x86 que obtuvo resultados similares.

Se ha observado a lo largo de la experimentación que la tasa de fotogramas procesando vídeo en este sistema nunca supera los 15FPS. Aunque esto puede sugerir que la Raspberry Pi no es capaz de ejecutar procesamiento de vídeo en tiempo real, es importante considerar que no todas las aplicaciones tienen los mismos requisitos, alcanzar 15 FPS es suficiente como para considerar la viabilidad de esta plataforma para solucionar problemas reales.

Aunque probablemente nunca se alcance el rendimiento deseado de 25 o 30 FPS por limitaciones hardware, los siguientes temas relativos a este trabajo siguen siendo interesantes.

- Procesamiento de vídeo en tiempo real en un sistema de memoria distribuida basado en Raspberry Pi. Esto puede ser interesante porque construir un clúster de Raspberry Pi es más barato que utilizando computadores de escritorio o servidores tradicionales además de tener un consumo energético más bajo.
- Procesamiento de vídeo en Computadores del tipo "System on a Chip" más potentes. Hay sistemas más potentes, con un consumo energético mayor y un precio ligeramente más alto que es posible que logren alcanzar 25 o 30 FPS.
- Optimizar código OpenCL para lograr los objetivos de tiempo real en otras plataformas. OpenCL no es muy conocido y puede resultar interesante portar algoritmos de procesamiento de vídeo más interesantes. También es importante denotar que OpenCL es muy especializado y tiene pocos usuarios en campos que no estén relacionados con el *High Performance Computing.*
- Procesamiento de vídeo en tiempo real, comparando CUDA y OpenCL. Muchos servidores ejecutan aplicaciones programadas en CUDA porque utilizan dispositivos de NVIDIA. Si la diferencia de rendimiento no es grande, puede ser conveniente empezar a migrar programas que utilicen CUDA a OpenCL puesto que CUDA solo soporta dispositivos de NVIDIA. Aunque actualmente esta empresa domina el mercado, en un futuro puede que surjan dispositivos que reemplacen a los de NVIDIA y esto requiere portar el código. Haber portado el código a OpenCL anticipándose a esto puede suponer una gran ventaja frente a competidores.

La Raspberry Pi, es una plataforma atractiva para una gran variedad de usos, el procesamiento de vídeo como se ha visto en este estudio, puede ser uno de ellos.

Por otra parte, este estudio permite resaltar la importancia de tener código portable en sistemas heterogéneos puesto que el código se puede reutilizar en otros sistemas. De hecho, la potencia de portabilidad de OpenCL se ha demostrado en la experimentación, habiendo compilado y ejecutado el mismo código en dos plataformas con arquitecturas diferentes.

Además, los sistemas heterogéneos siguen progresando y nuevos modelos más potentes son lanzados al mercado constantemente (Raspberry Pi 4, por ejemplo). Esto aumenta la importancia de tener un estándar como OpenCL que permite potencia y portabilidad para cualquier sistema que lo soporte.

Bibliografía

[1] SocialCompare, “Raspberry Pi model comparison,” [Online]. Available: https://socialcompare.com/es/comparison/raspberrypi-models-comparison. [Accessed 19 06 2020].

[2] "Vim3 | Khadas," [Online]. Available: https://www.khadas.com/vim3. [Accessed 24 06 2020].

[3] OrangePi, “OrangePi,” [Online]. Available: http://www.orangepi.org/. [Accessed 19 06 2020].

[4] Banana Pi, “Banana Pi,” [Online]. Available: http://www.banana-pi.org/. [Accessed 19 06 2020].

[5] elandroid, “Las 12 mejores android TV box de 2020,” El androide feliz, 17 02 2020. [Online]. Available: https://elandroidefeliz.com/mejores-android-tv-box-del-momento/. [Accessed 19 06 2020].

[6] A. Harju, T. Siro, F. Canova, S. Hakala and T. Rantalaiho, “Computational Physics on Graphics Processing Units,” Helsinki, 2012.

[7] NVIDIA, “Deep Learning, NVIDIA Developer,” [Online]. Available: https://developer.nvidia.com/deep-learning. [Accessed 23 06 2020].

[8] NVIDIA, “Cuda Toolkit documentation,” NVIDIA, [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. [Accessed 22 06 2020].

[9] S. Carrillo, J. Siegel and X. Li, “Impact analysis of conditional and loop statements for the NVIDIA G80 architecture,” *Ingeniería y Desarrollo,* no. 27, 2010.

[10] R. Nozal and J. L. Bosque, “EngineCL: Usability and Performance in Heterogeneous Computing,” in *Avances en Arquitectura y Tecnología de Computadores*, Cáceres, 2019.

[11] G. Fernández, “Poesía binaria,” 26 06 2011. [Online]. Available: https://poesiabinaria.net/2011/06/leyendo-archivos-de-imagen-en-formato-bmp-en-c/. [Accessed 19 06 2020].

[12] M. Carvalho, “Video Grabber example using libv4l,” [Online]. Available: https://www.linuxtv.org/downloads/v4l-dvb-apis-old/v4l2grab-example.html. [Accessed 19 06 2020].

[13] OpenCV, “OpenCV docs,” [Online]. Available: https://docs.opencv.org/2.4/index.html. [Accessed 19 06 2020].

[14] OpenCV, “Opencv repository,” [Online]. Available: https://github.com/opencv/opencv. [Accessed 19 06 2020].

[15] OpenCV, “OpenCV,” [Online]. Available: https://opencv.org/about/. [Accessed 24 03 2020].

[16] K. Group, “Khronos Group,” [Online]. Available: https://www.khronos.org/opencl/. [Accessed 03 24 2020].

[17] O. group, “OpenMP,” [Online]. Available: https://www.openmp.org/about/about-us/. [Accessed 24 03 2020].

[18] RipTutorial, “RipTutorial,” [Online]. Available: https://riptutorial.com/openmp/example/23425/addition-of-two-vectors-using-openmp-parallel-for-construct. [Accessed 15 06 2020].

[19] Microsoft, “Microsoft Docs,” Microsoft, 22 01 2019. [Online]. Available: https://docs.microsoft.com/es-es/cpp/parallel/openmp/d-using-the-schedule-clause?view=vs-2019. [Accessed 15 06 2020].

[20] Seaborn, “Seaborn,” 2020. [Online]. Available: https://seaborn.pydata.org/. [Accessed 04 June 2020].

[21] Pandas, “Pandas,” 28 May 2020. [Online]. Available: https://pandas.pydata.org/. [Accessed 04 June 2020].

[22] E. Elboher and M. Werman, “Efficient and Accurate Gaussian Image Filtering Using Running Sums,” in *International Conference on Intelligent Systems Design and Applications, ISDA,* 2011.

[23] K. &. T. K. Padmavathi, “Implementation of RGB and Grayscale Images in Plant Leaves Disease Detection - Comparative Study,” *Indian Journal of Science and Technology,* vol. 9, 2016.

[24] OpenCV, “Geometric Image Transformations - OpenCV docs,” [Online]. Available: https://docs.opencv.org/2.4/modules/imgproc/doc/geometric_transformations.html. [Accessed 06 26 2020].

[25] S. Mallick, «Learn OpenCV,» 11 03 2018. [En línea]. Available: https://www.learnopencv.com/tag/warpperspective/. [Último acceso: 06 26 2020].

[26] FFMPEG, “FFmpeg,” [Online]. Available: https://ffmpeg.org/. [Accessed 17 06 2020].

[27] Microsoft, “Bitmap Storage,” Microsoft, 31 05 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/gdi/bitmap-storage. [Accessed 10 06 2020].

[28] AMD, “OpenCL programming guide - ROCm documentation,” [Online]. Available: https://rocmdocs.amd.com/en/latest/Programming_Guides/Opencl-programming-guide.html. [Accessed 22 06 2020].

[29] Microsoft Docs, “Extensión SIMD - Microsoft Docs,” 20 03 2019. [Online]. Available: https://docs.microsoft.com/es-es/cpp/parallel/openmp/openmp-simd?view=vs-2019. [Accessed 17 06 2020].

[30] R. p. Foundation, «Raspberry pi,» [En línea]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. [Último acceso: 24 03 2020].

[31] E. Rosten, «CVD Projects,» [En línea]. Available: https://www.edwardrosten.com/cvd/. [Último acceso: 09 04 2020].

[32] M. C. Chehab, «LinuxTV,» [En línea]. Available: https://www.linuxtv.org/downloads/v4l-dvb-apis-old/v4l2grab-example.html. [Último acceso: 06 May 2020].

[33] D. Akgün, U. Sakoglu, M. Mete and B. Adinoff, “GPU-accelerated dynamic functional connectivity analysis for functional MRI data using OpenCL,” in *IEEE International Conference on Electro/Information Technology*, 2014.

[34] Hardkernel, “Odroid N2,” [Online]. Available: https://www.hardkernel.com/shop/odroid-n2-with-4gbyte-ram/. [Accessed 19 06 2020].

[35] Microsoft, “Microsoft Docs,” 12 05 2020. [Online]. Available: https://docs.microsoft.com/es-es/windows/wsl/about. [Accessed 16 06 2020].