Trabajo Fin de Grado en Ingeniería Electrónica Industrial

Procesado de vídeo en hardware utilizando sistemas SoC

Universidad de Sevilla

Escuela Politécnica Superior de Sevilla

Autor: Antonio Moral Villarín

Tutor: Carlos Jesús Jiménez Fernández

Departamento de Tecnología Electrónica

Fecha: Curso 2023-2024

RESUMEN

En la era actual, donde la cantidad de datos de vídeo generados y consumidos crece exponencialmente, la necesidad de procesamiento de vídeo eficiente y en tiempo real nunca ha sido más crítica. Este trabajo se centra en el uso de sistemas SoC (System on Chip) para el procesado de vídeo, destacando su capacidad para ofrecer soluciones compactas, energéticamente eficientes y de alto rendimiento. A través de la implementación y evaluación de algoritmos específicos de procesado de vídeo en sistemas SoC, este proyecto busca probar cómo estos sistemas pueden superar las limitaciones de las arquitecturas de hardware convencionales, ofreciendo una plataforma viable para aplicaciones de procesamiento de vídeo en tiempo real en diversos campos, desde la seguridad pública hasta la telemedicina y el entretenimiento.

Utilizando una metodología rigurosa que combina pruebas preliminares, desarrollo y pruebas tanto de software como de hardware, este trabajo explora la flexibilidad, eficiencia y capacidad de procesamiento de los sistemas SoC. Se comparan las implementaciones en software y hardware, evaluando el rendimiento, el consumo de energía y la eficiencia de procesamiento, a fin de establecer las ventajas significativas de los sistemas SoC sobre las soluciones tradicionales. Además, se discute la aplicabilidad de estos sistemas en el procesado de vídeo, resaltando su potencial para adaptarse a las necesidades cambiantes de la tecnología moderna y los requisitos de procesamiento.

Los resultados obtenidos indican que los sistemas SoC representan una evolución importante en el campo del procesado de vídeo, ofreciendo mejoras notables en términos de eficiencia energética y capacidad de procesamiento en tiempo real, lo que los convierte en una solución prometedora para futuras investigaciones y desarrollos tecnológicos en esta área. Este estudio no solo proporciona una base sólida para la comprensión de la aplicación de los sistemas SoC en el procesado de vídeo, sino que también abre el camino para futuras innovaciones en el diseño y aplicación de estas tecnologías en una amplia gama de aplicaciones prácticas.

ABSTRACT

In today's era, where the amount of video data generated and consumed is growing exponentially, the need for efficient, real-time video processing has never been more critical. This work focuses on the use of SoC (System on Chip) systems for video processing, highlighting their ability to offer compact, energy-efficient and high-performance solutions. Through the implementation and evaluation of specific video processing algorithms in SoC systems, this project seeks to demonstrate how these systems can overcome the limitations of conventional hardware architectures, offering a viable platform for real-time video processing applications in diverse fields, from public safety to telemedicine and entertainment.

Using a rigorous methodology that combines preliminary testing, development and testing of both software and hardware, this paper explores the flexibility, efficiency and processing power of SoC systems. Software and hardware implementations are compared, evaluating performance, power consumption and processing efficiency, in order to establish the significant advantages of SoC systems over traditional solutions. In addition, the applicability of these systems in video processing is discussed, highlighting their potential to adapt to the changing needs of modern technology and processing requirements.

The results obtained show that SoC systems represent an important evolution in the field of video processing, offering remarkable improvements in terms of energy efficiency and real-time processing capability, making them a promising solution for future research and technological development in this area. This study not only provides a solid foundation for understanding the application of SoC system in video processing, but also paves the way for future innovations in the design and application of these technologies in a wide range of practical applications.

ÍNDICE

1 INTRODUCCIÓN

1.1 Contextualización y justificación del proyecto

El procesado de vídeo, un campo crucial en la era digital, se ha convertido en un pilar en diversas aplicaciones que abarcan desde la seguridad pública y vigilancia hasta el entretenimiento y la telemedicina. Con el volumen de datos de vídeo generados diariamente alcanzando cifras sin precedentes, impulsado por el crecimiento exponencial de dispositivos móviles y cámaras IoT (Internet of Things), se hace cada vez más necesaria la búsqueda de soluciones eficientes y escalables para su procesamiento.

El procesado de vídeo no solo es crucial para convertir grandes volúmenes de datos brutos en información útil y accionable, sino que también juega un papel fundamental en la mejora de la calidad de vídeo, la detección y reconocimiento de patrones, y la reducción de la carga de datos a transmitir y almacenar. Estas capacidades son esenciales para áreas como:

- Seguridad y Vigilancia: Donde el procesamiento en tiempo real puede identificar amenazas potenciales y mejorar los tiempos de respuesta.

- Salud: Facilitando diagnósticos a distancia más precisos y monitoreo continuo de pacientes.

- Automoción: En el desarrollo de sistemas avanzados de asistencia al conductor y vehículos autónomos.

- Entretenimiento: Mejorando la experiencia del usuario mediante la personalización de contenidos y la entrega de medios de alta calidad.

Pese a su importancia, el procesado de vídeo enfrenta desafíos significativos, especialmente relacionados con la eficiencia energética, la necesidad de procesamiento en tiempo real y la gestión de la creciente demanda de resoluciones de vídeo más altas. Los métodos tradicionales, a menudo dependientes de infraestructuras de hardware voluminosas y consumo energético elevado, están cada vez más limitados por estas demandas.

Frente a estos desafíos, los sistemas SoC (System on Chip) emergen como una solución prometedora. Integrando todas las funcionalidades necesarias—procesadores, memoria, y periféricos—en un único microchip, los SoC ofrecen una plataforma compacta, energéticamente eficiente y de alto rendimiento para el procesado de vídeo. Este enfoque no solo permite superar las limitaciones de las arquitecturas convencionales, sino que también abre nuevas posibilidades para la innovación y la aplicación en campos emergentes.

Este proyecto se justifica por la necesidad de explorar y demostrar las capacidades de los sistemas SoC en el ámbito del procesado de vídeo. Al desarrollar e implementar algoritmos de procesado en un sistema SoC, se busca evidenciar su potencial para mejorar significativamente la eficiencia, el rendimiento y la escalabilidad de las aplicaciones de vídeo, marcando un camino hacia soluciones más sostenibles y adaptativas para los desafíos futuros en este campo dinámico y de rápida evolución.

1.2 Objetivos del TFG

La implementación de sistemas SoC en el procesado de vídeo abre un abanico de posibilidades y desafíos. A través de este Trabajo de Fin de Grado, se pretende no solo explorar estas posibilidades sino también contribuir de manera significativa al avance de la tecnología en este campo. A continuación, se detallan los objetivos que guiarán esta investigación:

1.2.1 Objetivo principal

El objetivo principal de este trabajo es demostrar la eficacia, eficiencia y versatilidad de los sistemas SoC-FPGA para el procesamiento de vídeo en tiempo real. Para ello se han utilizado, desarrollado e implementado Algoritmos de Procesado de Vídeo en Sistemas SoC. Esto implica la conceptualización, diseño, implementación y evaluación de algoritmos específicos que aprovechen la arquitectura integrada de los SoC-FPGA, destacando su capacidad para realizar tareas complejas de procesado de vídeo con un consumo energético y un espacio físico reducidos.

1.2.2 Objetivos secundarios

1. Aprender la estructura interna y las capacidades de los sistemas SoC-FPGA:

- Estudiar los recursos internos de los dispositivos SoC-FPGA, las posibilidades que ofrecen y la forma de interconexión entre la parte software y la parte hardware.

- Aprender la metodología de diseño necesaria para el desarrollo de este tipo de dispositivos, herramientas involucradas y flujo de diseño a seguir.

- Utilizar una plataforma concreta de SoC-FPGA, estudiando sus características y probando sobre ella los diseños realizados.

2. Comparar el Rendimiento de los Sistemas SoC con las Soluciones Tradicionales de Procesado de Vídeo:

- Evaluar la eficiencia energética, la velocidad de procesamiento y la capacidad de manejar flujos de datos de vídeo de alta resolución y frecuencia de

cuadros en comparación con soluciones basadas en arquitecturas de hardware convencionales.

- Analizar casos de uso específicos donde los SoC puedan ofrecer ventajas significativas sobre las plataformas de procesamiento tradicionales.

3. Demostrar la Aplicabilidad de los Sistemas SoC en Diversas Áreas:

- Identificar y desarrollar aplicaciones prácticas en campos como la seguridad, la salud y el entretenimiento, donde el procesado de vídeo juega un papel crucial.

- Explorar cómo la integración y eficiencia de los SoC pueden abrir nuevas vías para soluciones innovadoras en estos sectores.

4. Explorar la Flexibilidad de los Sistemas SoC para Adaptarse a Diferentes Algoritmos y Requisitos de Procesado:

- Investigar la capacidad de los SoC para ser reconfigurados y adaptarse a diferentes necesidades de procesamiento de vídeo, desde la detección de movimiento hasta el análisis de imagen avanzado.

- Evaluar el soporte de los SoC para el desarrollo rápido y la implementación de nuevos algoritmos de procesado de vídeo, facilitando la iteración y la innovación.

1.2.3 Justificación de los objetivos

Estos objetivos se justifican por la creciente demanda de soluciones de procesado de vídeo más eficientes y flexibles, capaces de adaptarse a las rápidas evoluciones tecnológicas y las cambiantes necesidades de aplicación. Al centrarse en los sistemas SoC, este TFG apunta a contribuir significativamente al avance de las tecnologías de procesado de vídeo, proporcionando evidencia empírica de sus ventajas y explorando su potencial para futuras innovaciones.

1.3 Estructura del documento

El documento se organiza en una serie de capítulos diseñados para guiar al lector a través del proceso completo de investigación, desde el contexto y la justificación del proyecto hasta las conclusiones finales y recomendaciones para futuros trabajos. A continuación, se presenta un desglose detallado de cada capítulo:

Capítulo 1: Introducción

- Contextualización y Justificación del Proyecto: Se introduce el tema del TFG, destacando la relevancia del procesado de vídeo en hardware utilizando sistemas SoC dentro del campo de la ingeniería electrónica industrial.

- Objetivos del TFG: Detallamiento específico de los objetivos que guían este trabajo, tanto generales como específicos, estableciendo el marco de lo que se espera lograr.

- Estructura del Documento: Presentación general del contenido de cada capítulo, ofreciendo al lector una guía de lo que encontrará en el documento.

Capítulo 2: Revisión Bibliográfica y Avances Tecnológicos

- Procesadores y SoCs en Procesado de Vídeo: Se profundiza en el papel que juegan los procesadores y los SoCs en el procesamiento de vídeo, cubriendo desde microcontroladores y microprocesadores hasta familias de FPGA y SoC.

- Metodología de Diseño en Procesado de Vídeo: Exploración de las herramientas y metodologías empleadas en el diseño para el procesado de vídeo, incluyendo el uso de herramientas de desarrollo y simulación.

Capítulo 3: Pruebas Preliminares

- Pruebas Iniciales y Configuración: Descripción de las configuraciones iniciales y pruebas preliminares realizadas para establecer un entorno de desarrollo adecuado.

- Selección y Justificación de las Pruebas Iniciales Realizadas: Explicación de cómo se eligieron las pruebas iniciales y por qué son relevantes para el proyecto.

- Conclusiones: Síntesis de los hallazgos y aprendizajes obtenidos de estas pruebas preliminares.

Capítulo 4: Pruebas Software

- Metodología de Pruebas: Explicación detallada de la metodología seguida para las pruebas de software, incluyendo selección de herramientas y criterios de comparación.

- Desarrollo de Pruebas Software: Exposición del proceso de desarrollo de las pruebas de software y análisis de los resultados obtenidos.

Capítulo 5: Pruebas Hardware

- Desarrollo de Filtros de Vídeo en Vitis HLS: Descripción del proceso de diseño y optimización de filtros de vídeo, así como su validación.

- Integración de IP en Diseño Vivado: Detalles sobre la integración de IP en el diseño usando Vivado y las pruebas de rendimiento realizadas.

- Implementación y Pruebas en Jupyter: Explicación de cómo se utilizaron los cuadernos Jupyter para implementar y probar los diseños en hardware.

Capítulo 6: Comparación y Análisis de Prestaciones

- Evaluación de Tiempos de Procesamiento y Uso de Recursos: Comparación entre implementaciones en software y hardware, evaluando eficiencia y uso de recursos.

- Análisis de Eficiencia Entre Software y Hardware: Discusión sobre las ventajas y desventajas de cada enfoque basado en los resultados de las pruebas.

Capítulo 7: Conclusiones y Trabajo Futuro

- Conclusiones Generales del Proyecto: Resumen de los objetivos alcanzados, las contribuciones principales y una comparación de prestaciones entre software y hardware.

- Limitaciones y Retos Encontrados: Reflexión sobre los desafíos técnicos y limitaciones del estudio.

- Propuestas para Futuras Investigaciones: Sugerencias para investigaciones futuras basadas en los hallazgos y experiencias del proyecto.

Con el objetivo de facilitar toda la información y ayuda posible a quienes les interese replicar y seguir investigando, todos los archivos relacionados con el proyecto están disponibles en mi repositorio de GitHub. En este repositorio, se encuentra la documentación completa, el código fuente, los datos utilizados y cualquier otro material relevante que he desarrollado a lo largo de mi investigación y desarrollo.

El objetivo al poner estos recursos a disposición del público es fomentar la transparencia, facilitar el aprendizaje y promover la colaboración. Espero que encuentren útil el contenido y que sirva como una valiosa referencia para futuros proyectos académicos y profesionales.

Para acceder a todos los archivos del TFG, visitar mi repositorio de GitHub en el siguiente enlace: https://github.com/Antoniomv7/Hardware-video-processing-using-SoC-systems

Agradezco de antemano cualquier comentario, sugerencia o contribución que deseen hacer.

2 REVISIÓN BIBLIOGRÁFICA Y AVANCES TECNOLÓGICOS

En este capítulo se abordan los conocimientos previos esenciales para la ejecución del proyecto y, por ende, este Trabajo de Fin de Grado. Se explora el contexto en el que se ha gestado y el papel que desempeña en la actualidad la tecnología SoC FPGA en el panorama contemporáneo.

Aunque gran parte de la información que aquí se describe es válida para cualquier aplicación, nos vamos a centrar en aplicaciones de vídeo porque es el ámbito sobre el que se ha desarrollado este trabajo.

2.1 Procesadores y SoCs en procesado de vídeo

El procesamiento de vídeo ha experimentado una evolución notable impulsada por los avances en microcontroladores y microprocesadores. Esta sección explora cómo estos componentes han cambiado el panorama del procesado de vídeo, desde sus primeros usos has las aplicaciones modernas.

2.1.1 Microcontrolador, definición y características

Un microcontrolador es un dispositivo compacto que integra todos los componentes necesarios de un computador en un solo chip. Incluye un procesador (CPU[1]), memoria (RAM[2] y ROM[3]), y periféricos de entrada/salida, todo en un mismo circuito integrado.

Debido a su diseño integrado y eficiencia energética, los microcontroladores son ideales para aplicaciones de control en sistemas embebidos, como electrodomésticos, automóviles, y dispositivos de control industrial. Algunas de

[1] CPU: Unidad Central de Procesamiento, es el componente principal de un dispositivo electrónico que ejecuta instrucciones de programas y realiza operaciones aritméticas, lógicas y de control básicas. Es responsable de interpretar y ejecutar las instrucciones del software, así como de coordinar y controlar todas las operaciones del sistema.

[2] RAM: Memoria de Acceso Aleatorio, es un tipo de memoria volátil que se utiliza en los ordenadores y otros dispositivos electrónicos para almacenar datos y programas en uso temporalmente. Es una forma de almacenamiento de acceso rápido que permite a la CPU acceder rápidamente a los datos que necesita para realizar operaciones. Sin embargo, la RAM es volátil, lo que significa que los datos se pierden cuando se apaga la energía, por lo que se utiliza principalmente para almacenar datos temporales durante la sesión de uso del dispositivo.

[3] ROM: Memoria de Solo Lectura, es un tipo de memoria no volátil que almacena datos permanentemente y no se borran cuando se apaga la energía. Contiene información esencial para el funcionamiento básico del sistema, como el firmware del dispositivo, el código de arranque del sistema operativo y otros programas esenciales. A diferencia de la RAM, la ROM no se puede modificar o reescribir fácilmente después de ser fabricada. Se utiliza para almacenar datos que no cambian con el tiempo y que son necesarios para iniciar y operar el dispositivo de forma básica.

las familias más conocidas incluyen AVR (utilizado en Arduino), PIC de Microchip, y ARM Cortex-M (como la familia de microcontroladores STM32).

2.1.2 Microprocesador, definición y características

El microprocesador, a menudo referido simplemente como procesador, es el cerebro de un computador y se encarga de ejecutar las instrucciones del programa. A diferencia de los microcontroladores, los microprocesadores requieren componentes externos adicionales, como memoria y periféricos de entrada/salida, para funcionar.

Los microprocesadores son ampliamente utilizados en ordenadores personales, servidores, teléfonos móviles y en equipos donde se requiere mayor capacidad de procesamiento. Entre las familias de microprocesadores más populares se encuentran Intel Core, AMD Ryzen, y ARM Cortex-A. Ejemplos concretos son Intel Core i7 (utilizado en PCs y portátiles), Apple M1 (basado en ARM, utilizado en MacBooks y iPads).

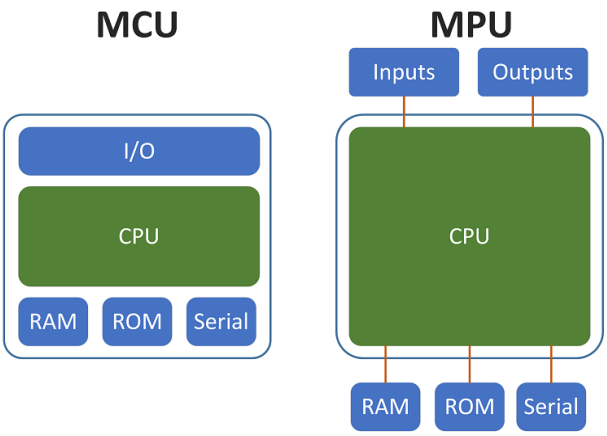

Figura 1 – Arquitectura básica de un Microcontrolador (MCU) vs Microprocesador (MPU). Extraída de [1]

2.1.3 Diferencias clave

Los microcontroladores son sistemas integrados diseñados para tareas específicas y control, mientras que los microprocesadores son más flexibles y potentes, diseñados para una amplia gama de aplicaciones informáticas.

Los microcontroladores generalmente son más económicos y menos complejos que los microprocesadores debido a su enfoque en tareas específicas.

Los microprocesadores suelen tener mayor capacidad de procesamiento y memoria, adecuados para aplicaciones que requieren manejar grandes cantidades de datos o realizar tareas complejas.

2.1.4 Historia y evolución de microcontroladores y microprocesadores en el contexto de procesamiento de vídeo

Los primeros microprocesadores, con limitada capacidad de cómputo, estaban enfocados en tareas básicas de computación. Los microcontroladores, por su parte, diseñados para controlar funciones específicas en dispositivos electrónicos, tenían un papel secundario en el procesamiento de vídeo debido a su limitada capacidad de procesamiento y memoria.

Con el advenimiento de microprocesadores más potentes y la miniaturización de componentes, se produjo un salto significativo en el procesamiento de vídeo. La integración de capacidades de procesamiento gráfico en los microprocesadores permitió manejar tareas de vídeo más complejas, como la codificación y decodificación de alta definición.

En la actualidad, los microprocesadores no solo gestionan tareas generales de computación, sino que también incorporan unidades de procesamiento gráfico (GPU) y capacidades de procesamiento paralelo, facilitando así tareas intensivas de procesamiento de vídeo como la renderización 3D, realidad virtual y vídeo en 4K o superior.

Un aspecto interesante a destacar son las dos opciones más importantes en cuanto al número de instrucciones. Actualmente existen dos opciones diferenciadas, la RISC y la CISC. La arquitectura RISC (Reduced Instruction Set Computing), es una arquitectura de diseño de CPU que se caracteriza por tener un conjunto de instrucciones reducido y simples. La arquitectura RISC tiende a simplificar el diseño de la CPU y facilita la optimización del hardware, lo que resulta en un mejor rendimiento y menor consumo de energía en comparación con las arquitecturas de CPU CISC. La arquitectura CISC (Complex Instruction Set Computing), es una arquitectura de diseño de CPU que se caracteriza por tener un conjunto de instrucciones más complejo y variado. Esto permite que los programas se escriban en menos líneas de código, pero puede resultar en un rendimiento menos eficiente y más impredecible en comparación con la arquitectura RISC.

Desde el punto de vista de aplicaciones de video, las arquitectura RISC y CISC ofrecen enfoques distintos en estas aplicaciones. Los microprocesadores RISC, con instrucciones más simples y un enfoque en la eficiencia del cómputo, pueden ser más adecuados para aplicaciones de procesamiento de video en tiempo real. Por otro lado, los CISC, con un conjunto de instrucciones más amplio y complejo, pueden ofrecer ventajas en tareas de procesamiento de video que requieren una mayor variedad de operaciones computacionales.

2.1.5 Ejemplos relevantes en la industria

Sistemas de vigilancia inteligente

La vigilancia de seguridad ha evolucionado desde cámaras analógicas simples a sistemas inteligentes capaces de análisis de vídeo en tiempo real.

Un ejemplo de microprocesador específico utilizado en este campo podría ser el Intel Core i7, que permite el procesamiento de vídeo de alta resolución y el análisis de imágenes en tiempo real. Estos sistemas utilizan algoritmos avanzados para reconocimiento facial, detección de movimiento y análisis conductual. El procesamiento de vídeo en tiempo real requiere una gran capacidad de cómputo, que es proporcionada por los microprocesadores avanzados.

El impacto de las innovaciones tecnológicas en la seguridad y la capacidad de respuesta ante situaciones críticas ha sido profundamente transformador. Estos avances han permitido una mejora significativa en ambos campos, contribuyendo a una gestión más eficaz y eficiente de emergencias y crisis. La integración de tecnologías avanzadas, como la inteligencia artificial, el análisis de big data y la conectividad en tiempo real, ha revolucionado los protocolos de seguridad y respuesta a incidentes.

Gracias a estas tecnologías, las autoridades y organizaciones pueden anticiparse a situaciones potencialmente peligrosas, identificar riesgos con mayor precisión y desplegar recursos de manera más estratégica. La capacidad para analizar grandes volúmenes de datos en tiempo real facilita la toma de decisiones informadas, permitiendo una actuación rápida y coordinada que puede salvar vidas y minimizar daños.

Smartphones con capacidades avanzadas de vídeo

Los smartphones modernos han revolucionado la forma en que capturamos y compartimos vídeos. Un ejemplo relevante es el Qualcomm Snapdragon 865, que ofrece capacidades avanzadas de procesamiento de vídeo, incluyendo la grabación en 4K y funciones de inteligencia artificial.

Estos dispositivos utilizan microprocesadores para procesar algoritmos de estabilización de imagen, mejorar la calidad de vídeo en condiciones de baja luz, y permitir funciones avanzadas como el zoom digital de alta resolución.

La revolución tecnológica ha transformado radicalmente la comunicación móvil y el entretenimiento, otorgando a los usuarios el poder de capturar y compartir contenido de vídeo de alta calidad con una facilidad sin precedentes. La disponibilidad de dispositivos móviles avanzados ha democratizado la creación de contenido audiovisual, permitiendo a cualquier persona con un teléfono inteligente convertirse en creador. Este cambio ha eliminado las barreras entre

creadores y espectadores, fomentando una cultura de participación activa y dando lugar a nuevos géneros y formatos en el entretenimiento digital.

Consolas de videojuegos de alta definición

Las consolas de videojuegos modernas proporcionan experiencias de juego inmersivas con gráficos de alta resolución. La PlayStation 5 de Sony, por ejemplo, utiliza un procesador personalizado basado en la arquitectura AMD Zen 2, capaz de procesar gráficos complejos y efectos de vídeo en tiempo real. Estas consolas utilizan microprocesadores avanzados para renderizar gráficos en 3D, procesar físicas de juego realistas y ofrecer soporte para juegos en 4K.

La evolución tecnológica en el ámbito de los videojuegos ha enriquecido enormemente la experiencia de juego, introduciendo gráficos de alta fidelidad y una experiencia de juego fluida que elevan el entretenimiento doméstico a niveles sin precedentes. Las innovaciones en renderización, inteligencia artificial y físicas han creado mundos virtuales detallados y realistas, ofreciendo una inmersión y respuesta al jugador mejoradas. Asimismo, el avance en el hardware ha permitido una ejecución más eficiente, reduciendo los tiempos de carga y facilitando una experiencia de juego continua y envolvente. Estos progresos no solo han redefinido lo que los jugadores esperan de los videojuegos, sino que también han marcado el inicio de una nueva era en el ocio digital.

2.1.6 Tecnologías emergentes: Inteligencia Artificial y aprendizaje automático en procesamiento de vídeo

Microprocesadores con capacidades de Inteligencia Artificial y Machine Learning integradas

Los microprocesadores contemporáneos están integrando capacidades de Inteligencia Artificial (IA) y Aprendizaje Automático, lo que posibilita un procesamiento de vídeo más inteligente y adaptable. Esta integración facilita funciones avanzadas tales como el reconocimiento de patrones en vídeo, la optimización automática de la calidad de imagen y el procesamiento de vídeo en tiempo real, especialmente en aplicaciones como la vigilancia inteligente.

Se anticipa que la inclusión de la IA en los microprocesadores conduzca a avances significativos en áreas como la realidad aumentada y la personalización del contenido de vídeo basada en el aprendizaje de las preferencias del usuario.

Los microprocesadores actuales están cada vez más equipados con núcleos dedicados o módulos específicos para tareas de IA y Machine Learning (ML). Estos procesadores se están empleando en una amplia gama de aplicaciones en el procesamiento de vídeo, que van desde la mejora automática de la calidad de imagen hasta el reconocimiento y seguimiento de objetos, así como la generación de efectos visuales mediante técnicas de IA.

Algunos ejemplos relevantes son el NVIDIA Tegra X1, utilizado en sistemas de entretenimiento y automóviles autónomos, o el Google Edge TPU, diseñado para llevar capacidades de ML a dispositivos de borde, demuestran esta integración.

Redes neuronales y aprendizaje profundo en el procesado de vídeo

La implementación de redes neuronales y técnicas de aprendizaje profundo está transformando el procesamiento de vídeo, permitiendo un análisis y una generación de contenido más sofisticados. Estas tecnologías permiten el análisis detallado de contenido de vídeo, reconocimiento de patrones y generación de vídeo automatizada. Son clave en aplicaciones como la edición de vídeo inteligente, la restauración de vídeo y la creación de contenido sintético.

Se espera que estas tecnologías permitan avances en áreas como la generación automática de resúmenes de vídeo, la traducción de lenguaje de signos en tiempo real y la creación de efectos visuales hiperrealistas en películas y videojuegos.

Optimización de hardware para Machine Learning en procesado de vídeo

El procesamiento de vídeo con ML requiere una gran cantidad de recursos computacionales. Por ello, se están desarrollando microprocesadores específicamente optimizados para estas tareas. Las innovaciones incluyen la implementación de TPUs[4] para manejar eficientemente los cálculos matriciales de las redes neuronales, y el desarrollo de arquitecturas de procesadores que pueden realizar operaciones de ML de manera más eficiente en términos de energía y tiempo.

Estos avances están permitiendo aplicaciones de ML en tiempo real para el procesamiento de vídeo en dispositivos móviles, drones, cámaras de seguridad y otros sistemas embebidos.

Desafíos y consideraciones futuras

Incluyen la gestión del calor, el consumo de energía y la necesidad de equilibrar la potencia de procesamiento con la eficiencia energética. Con el aumento en la capacidad de análisis de vídeo, surgen preocupaciones sobre la privacidad y el

[4] TPU: Tensor Processing Unit, es un tipo de unidad de procesamiento diseñada específicamente para realizar operaciones de multiplicación y acumulación de matrices, que son fundamentales en el entrenamiento y la ejecución de modelos de inteligencia artificial, especialmente redes neuronales profundas. Las TPUs están optimizadas para trabajar con grandes cantidades de datos y realizar cálculos matriciales de manera eficiente y rápida, lo que las hace ideales para aplicaciones de aprendizaje automático y procesamiento de datos a gran escala. Son utilizadas por empresas como Google en sus servicios de inteligencia artificial y computación en la nube para acelerar el procesamiento de datos y mejorar el rendimiento de los modelos de aprendizaje automático.

uso ético de la tecnología de reconocimiento facial y de análisis de comportamiento.

Se prevé un avance continuo en la integración de IA y ML en microprocesadores, lo que conducirá a una mayor personalización y sofisticación en el procesamiento de vídeo, así como a nuevos desafíos y oportunidades.

2.2 Familias de FPGA y SoC FPGA

2.2.1 Introducción FPGA y SoC: definiciones y diferencias clave

FPGA

Una FPGA (Field-Programmable Gate Array) es un dispositivo semiconductor que contiene bloques de lógica cuya interconexión y funcionalidad pueden ser configuradas por el usuario después de la fabricación. Esto permite una gran flexibilidad en el diseño de hardware.

Entre sus principales características destacan una alta flexibilidad, su capacidad de reconfiguración y aptitud para prototipado rápido y pruebas de concepto.

Los SoC FPGA representan una combinación innovadora de tecnología FPGA y la arquitectura de un SoC, brindando una solución versátil y potente para el diseño de sistemas embebidos. Estos dispositivos integran las características programables de un FPGA con los bloques de un SoC, incluyendo procesadores, interfaces de comunicación, y otras funcionalidades. Esta integración permite a los diseñadores desarrollar sistemas altamente personalizables y con capacidad de reconfiguración en tiempo real, lo cual es ideal para aplicaciones que requieren una gran flexibilidad y rendimiento. A continuación, se describen los elementos básicos de los SoC FPGA, sus Configurable Logic Blocks (CLBs), y sus redes de interconexión:

- **Núcleos de Procesador**: Un SoC FPGA típicamente incorpora uno o más núcleos de procesador, los cuales pueden ser de arquitectura ARM o de otro tipo. Estos procesadores ejecutan el software del sistema y gestionan las tareas de alto nivel.

- **Bloques de Lógica Programable (CLBs)**: Son el corazón de un FPGA, proporcionando la lógica programable para implementar funciones digitales. Los CLBs pueden ser configurados para realizar una amplia gama de tareas lógicas, desde operaciones aritméticas básicas hasta la implementación de máquinas de estados complejas. Un CLB típicamente contiene:

    - **LUTs (Look-Up Tables)**: Utilizadas para implementar funciones lógicas. Una LUT puede ser configurada para realizar cualquier función lógica de sus entradas.

  - **Flip-Flops**: Proporcionan capacidad de almacenamiento temporal para implementar registros, contadores y otros elementos secuenciales.

  - **Multiplexores y Rutas de Conexión**: Permiten la selección de señales y la conexión entre diferentes partes del CLB.

- **Bloques de Memoria**: Los SoC FPGA contienen bloques de memoria RAM integrados que pueden ser utilizados para almacenamiento de datos temporal o como memoria para los procesadores.

- **Interfaces de Comunicación**: Incluyen una variedad de interfaces como Ethernet, USB, SPI, I2C, entre otras, para permitir la comunicación con otros dispositivos y sistemas.

- **Periféricos y Bloques IP**: Muchos SoC FPGA vienen con una gama de periféricos integrados y bloques de propiedad intelectual (IP) para funciones específicas, como controladores DMA, temporizadores, y unidades de gestión de energía.

Las redes de interconexión en un SoC FPGA juegan un papel crucial, ya que conectan los CLBs, los bloques de memoria, los procesadores, y los periféricos. Estas redes permiten la configuración de rutas de señales flexibles y eficientes a través del dispositivo, lo cual es esencial para la implementación de diseños complejos. Las redes de interconexión pueden ser estáticas o parcialmente reconfigurables, ofreciendo un balance entre la flexibilidad y el rendimiento.

SoC (System on Chip)

Un SoC (System on Chip) integra todos los componentes necesarios de un sistema electrónico en un solo chip, de ahí su nombre. Esto incluye la CPU, la GPU, la memoria, los periféricos de E/S y, en algunos casos, elementos específicos como un FPGA. En la Figura 2 se muestra un diagrama de bloques básico.

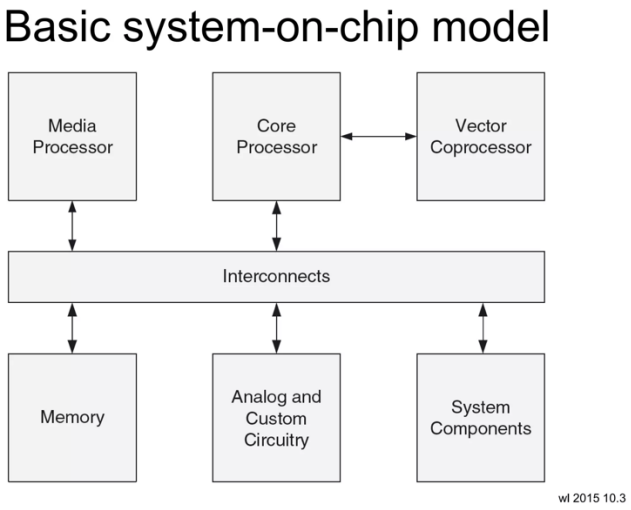

Figura 2 - Diagrama de bloques de un SoC básico. Extraída de [2]

Sus características principales son la integración completa del sistema, eficiencia energética y lo compacto de su implementación.

Diferencias clave

Mientras que los FPGAs ofrecen una mayor flexibilidad y reconfigurabilidad, los SoCs proporcionan una solución integrada y compacta. Los FPGAs son ideales para el desarrollo y la iteración rápida de sistemas de altas prestaciones, mientras que los SoCs son más adecuados para la producción a gran escala.

2.2.2 Ventajas de FPGA/SoC en el procesado de vídeo: flexibilidad, personalización y eficiencia energética

FPGA

La reconfigurabilidad inherente de los FPGAs los convierte en opciones idóneas para aplicaciones de procesamiento de vídeo que demandan personalización específica y la capacidad de adaptarse a diversos estándares y resoluciones. Estos dispositivos son altamente versátiles y pueden ajustarse para cumplir con los requisitos precisos de una amplia gama de escenarios de procesamiento de vídeo, desde sistemas de vigilancia hasta aplicaciones médicas y de entretenimiento.

Además de su capacidad de adaptación, las FPGAs destacan por su eficiencia en el manejo de tareas paralelas, lo cual resulta fundamental en el procesamiento de vídeo. En este sentido, pueden ejecutar de manera eficaz operaciones simultáneas como la codificación, decodificación y filtrado en tiempo real. Esta capacidad paralela no solo acelera el rendimiento del procesamiento de vídeo, sino que también permite la implementación de algoritmos más complejos y avanzados, como el procesamiento de imágenes de alta resolución y la detección de objetos en tiempo real.

Por otro lado, la naturaleza reconfigurable de los FPGAs facilita la actualización y adaptación de los sistemas de procesamiento de vídeo a medida que evolucionan los estándares y requisitos del mercado. Esto significa que los diseñadores pueden implementar cambios y mejoras en el hardware de manera rápida y flexible, sin necesidad de recurrir a costosas actualizaciones de hardware o rediseños completos.

En resumen, la combinación de reconfigurabilidad y capacidad de procesamiento paralelo hace que los FPGAs sean una opción poderosa y altamente efectiva para una amplia variedad de aplicaciones de procesamiento de vídeo, ofreciendo la flexibilidad y el rendimiento necesarios para abordar los desafíos actuales y futuros en este campo en constante evolución.

SoC

La tecnología de SoC reconfigurables presentan dos características distintivas que los separan de otros diseños de SoC. La primera radica en la capacidad de modificar la funcionalidad del hardware simplemente alterando el código encargado de la inicialización del sistema. La segunda característica deriva de esta capacidad reconfigurable. La reconfiguración puede llevarse a cabo un número ilimitado de veces después de la inicialización del hardware, lo que permite que múltiples aplicaciones aprovechen esta flexibilidad. Por ejemplo, un reproductor multimedia podría ajustar un códec de hardware[5] según el tipo de flujo de datos utilizado. Del mismo modo, un cliente o servidor de una Red Privada Virtual (VPN) podría emplear diferentes métodos de cifrado implementados en el hardware.

Los sistemas integrados son inherentemente complejos. Las partes de hardware y software de un diseño integrado constituyen proyectos por sí mismos. La fusión de estos dos componentes para que funcionen como un solo sistema presenta desafíos adicionales. Si a esto se agrega un proyecto de diseño de Field Programmable Gate Array (FPGA), la situación puede volverse considerablemente más complicada.

Para simplificar este proceso de diseño, Xilinx proporciona varios conjuntos de herramientas que requieren familiarización y siguen un flujo de desarrollo específico. Entre estas herramientas se encuentra el Entorno de Desarrollo Integrado (IDE) de Vivado, que incluye la herramienta IP Integrator, utilizada para integrar diseños basados en procesadores. Esta herramienta, combinada con la

[5] Códec de Hardware: Un códec de hardware es un dispositivo electrónico que se utiliza para codificar y decodificar datos de audio, vídeo o señales de comunicación de manera eficiente y en tiempo real. Está diseñado para realizar estas operaciones utilizando circuitos dedicados y optimizados en lugar de depender completamente del software o del procesador principal de un sistema. Los códecs de hardware son comunes en dispositivos como cámaras de vídeo, teléfonos móviles, sistemas de videovigilancia y equipos de comunicaciones, donde se requiere un procesamiento rápido y eficiente de datos multimedia. Su objetivo es mejorar el rendimiento y reducir la carga de trabajo del procesador principal, lo que permite una reproducción y grabación de audio y vídeo más fluida y de mayor calidad.

plataforma de software Xilinx Vitis HLS, ofrece un entorno integrado para el diseño y depuración de sistemas basados en microprocesadores y aplicaciones de software integrado.

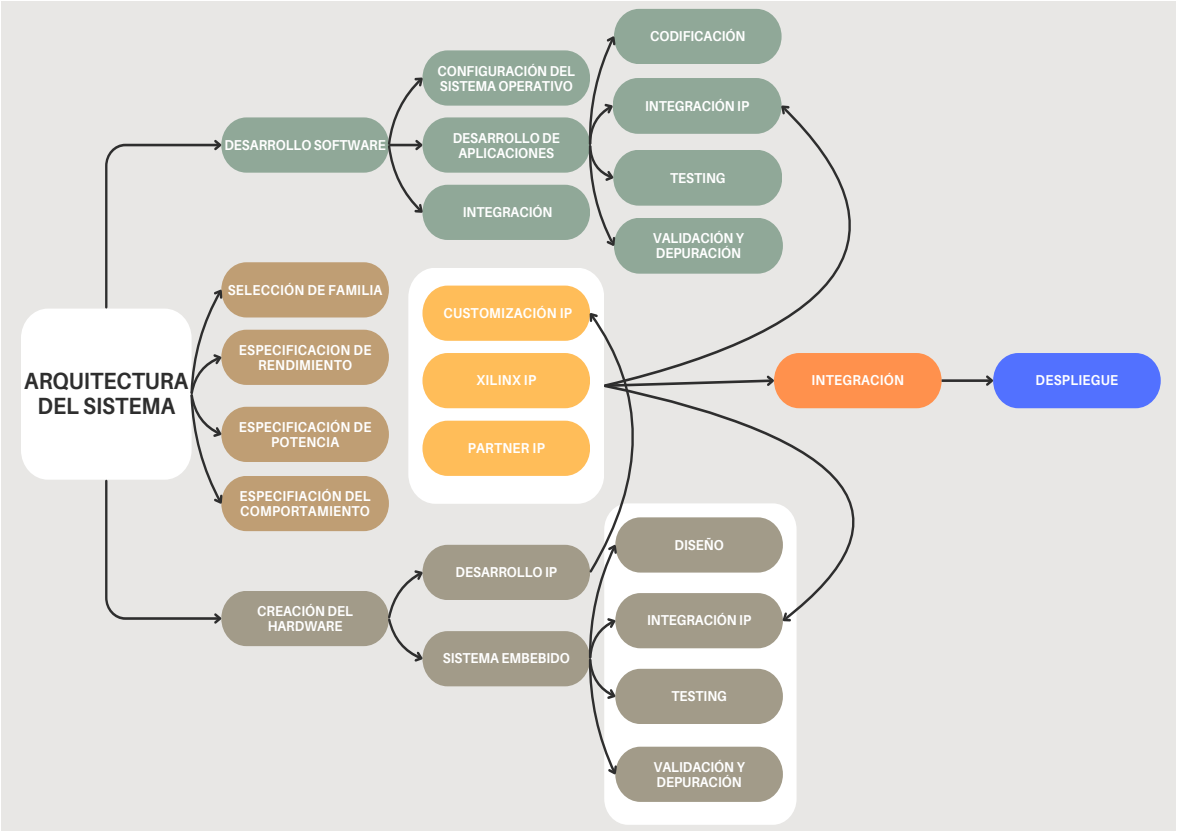

Figura 3 - Flujo de diseño de un sistema embebido. Extraída de [3]

2.3 Metodología de diseño en procesado de vídeo

Centraremos nuestra atención en las herramientas proporcionadas por Xilinx. Esta elección no es arbitraria; Xilinx es ampliamente reconocida por su liderazgo e innovación en el campo de los SoC FPGA, ofreciendo un ecosistema robusto de herramientas de diseño que facilitan desde la síntesis de alto nivel hasta la implementación final de los sistemas.

En este contexto, las herramientas de Xilinx como Vivado, para diseño y síntesis, y Vitis HLS, para aceleración y desarrollo de aplicaciones, se convierten en elementos centrales de nuestro trabajo. La plataforma PYNQ-Z2, específicamente, será nuestra base para la implementación, permitiendo un desarrollo ágil y eficiente gracias a su arquitectura que integra capacidades de procesamiento general a través de los núcleos ARM y la flexibilidad de la lógica programable FPGA.

La decisión de centrarnos en las herramientas de Xilinx radica en su capacidad para proporcionar una solución integral que abarca la parte de hardware y de

software, permitiendo explotar al máximo las ventajas que ofrecen los SoC FPGA.

2.3.1 Herramientas de desarrollo y simulación

2.3.1.1 Vitis HLS

Antes de explicar esta herramienta, es necesario conocer en qué consiste la Síntesis de Alto Nivel (High-Level Synthesis, HLS), ya que simplifica significativamente el proceso de diseño de hardware al permitir a los ingenieros expresar sus algoritmos y funcionalidades en lenguajes de programación de alto nivel que son más familiares y de más alto nivel de abstracción que los lenguajes de descripción de hardware tradicionales, como VHDL o Verilog.

El proceso de diseño con HLS implica varios pasos:

1. Captura de la especificación: Se define la funcionalidad deseada del hardware utilizando un lenguaje de programación de alto nivel.

2. Análisis y optimización: El código de alto nivel se analiza para identificar oportunidades de paralelismo, optimización de recursos y otras mejoras de rendimiento.

3. Mapeo a nivel de arquitectura: El código optimizado se mapea a elementos de hardware específicos, como unidades de procesamiento, bloques de memoria y conexiones de E/S.

4. Generación de código RTL: Se genera el código de descripción de hardware (RTL, Register Transfer Level) en un lenguaje de descripción de hardware tradicional, como VHDL o Verilog, a partir del código de alto nivel optimizado.

5. Síntesis y optimización de RTL: El código RTL generado se sintetiza en un dispositivo de hardware físico y se optimiza para cumplir con los requisitos de rendimiento, área y consumo de energía.

El proceso de diseño con HLS ofrece varios beneficios, incluyendo una mayor productividad de diseño, una reducción en el tiempo de desarrollo, una mayor portabilidad del código y una mejor verificación funcional. Sin embargo, también presenta desafíos, como la necesidad de una cuidadosa optimización del código de alto nivel para obtener un diseño de hardware eficiente, y la dependencia de herramientas de síntesis de alta calidad para producir resultados óptimos.

Vitis HLS es una herramienta poderosa y versátil para el desarrollo de algoritmos FPGA complejos, especialmente utilizada en el contexto de los sistemas embebidos y el procesado de vídeo en hardware. Esta herramienta, parte de la plataforma de software unificado de Vitis de AMD, facilita la creación de estos

algoritmos al permitir la síntesis de funciones escritas en C/C++ en RTL (Register Transfer Level) [4][5].

La integración estrecha de Vitis HLS con Vivado Design Suite y la plataforma de software unificado de Vitis es una de sus características destacadas. Esto permite un flujo de trabajo más fluido y una mayor eficiencia en el diseño de sistemas heterogéneos que incluyen lógica programable y procesadores ARM, entre otros componentes.

Entre las capacidades de Vitis HLS, se incluyen:

- Modelado de Implementación Paralela: Vitis HLS soporta construcciones de programación paralela que facilitan el modelado de implementaciones deseadas. Esto incluye tareas HLS para la concurrencia a nivel de proceso, vectores HLS para paralelismo a nivel de datos y flujos HLS para la comunicación entre tareas concurrentes.

- Conversión de C a RTL: La herramienta sintetiza diferentes partes del código C de manera distinta, como argumentos de funciones de nivel superior en puertos de E/S RTL y bucles de funciones C que se mantienen enrollados o en pipelines para mejorar el rendimiento.

- Simulación y Verificación: Vitis HLS ofrece flujos de simulación integrados para tiempos de verificación más rápidos, utilizando simulación C para validar la funcionalidad del código C y co-simulación C/RTL para verificar que el RTL generado sea funcionalmente idéntico al código fuente en C.

Estas características hacen que Vitis HLS sea una herramienta clave en el flujo de desarrollo de aceleración de aplicaciones de Vitis, compilando código C/C++ y OpenCL en un kernel para aceleración en la región de lógica programable (PL) de los dispositivos Xilinx [6].

Para un desarrollo exitoso en Vitis HLS, es crucial familiarizarse con sus capacidades, metodologías y flujos de trabajo específicos. Esto incluye el uso efectivo de directivas de compilador y la comprensión de las opciones de optimización y depuración disponibles. La documentación y tutoriales de Vitis HLS proporcionan una guía detallada y ejemplos prácticos para ayudar en este proceso, asimismo el Anexo Tutorial a este documento explica lo necesario para poder desarrollar e implementar los diseños creados en este proyecto.

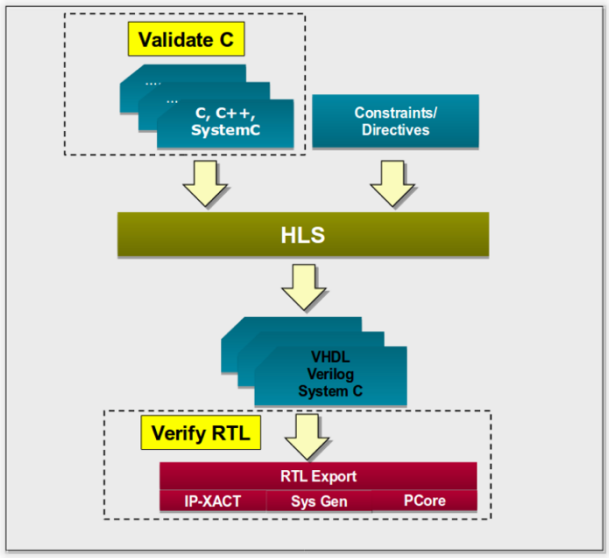

Figura 4 - Proceso de Síntesis de alto nivel. Imagen extraída de [7]

2.3.1.2 Vivado

Vivado es un software de diseño integral para SoCs adaptativos y FPGAs de Xilinx. Este software incluye una serie de herramientas y funcionalidades esenciales para el diseño de hardware, como la entrada de diseño, síntesis, colocación y rutado, y herramientas de verificación/simulación [8].

Las características clave de Vivado son:

1. Facilitación de Iteraciones de Diseño Más Rápidas: Vivado tiene tecnologías como compilación incremental[6] y Abstract Shell[7] que reducen significativamente los tiempos de iteración de diseño. Esto es especialmente útil cuando los cambios en el diseño son incrementales y se concentran en una pequeña parte del diseño [9].

2. Estimación Precisa del Consumo de Potencia: Vivado incluye el Power Design Manager, una herramienta avanzada para la estimación de la potencia, crucial para la toma de decisiones en el diseño de SoCs adaptativos y FPGAs. Esta herramienta es particularmente útil para dispositivos grandes

[6] Compilación Incremental: La compilación incremental es un proceso en el desarrollo de software donde solo se compilan las partes del código que han cambiado desde la última compilación, en lugar de compilar todo el proyecto nuevamente. Esto acelera el tiempo de compilación y optimiza los recursos del sistema, ya que solo se procesan las modificaciones realizadas, lo que resulta en un flujo de trabajo más eficiente y rápido.

[7] Abstract Shell: Puede entenderse como una abstracción o interfaz de línea de comandos que permite interactuar con un sistema informático de manera abstracta, es decir, sin necesidad de conocer los detalles específicos de su funcionamiento interno. Esto puede referirse a un tipo de shell o entorno de línea de comandos que proporciona comandos y funciones de más alto nivel para realizar tareas específicas, ocultando la complejidad subyacente del sistema operativo o del entorno de ejecución.

y complejos, proporcionando estimaciones precisas de potencia tempranas en el proceso de diseño.

3. Logro de Objetivos FMAX: Vivado proporciona características únicas como Report QoR Assessment (RQA), Report QoR Suggestions (RQS) y Intelligent Design Runs (IDR), que ayudan a alcanzar los objetivos de rendimiento de diseño de alta velocidad. Estas funcionalidades facilitan la convergencia hacia los objetivos de rendimiento en días en lugar de semanas.

Además, Vivado ofrece dos ediciones principales:

- Vivado ML Standard Edition: Una versión gratuita del software que proporciona acceso instantáneo a algunas funcionalidades y características básicas de Vivado, utilizada en este proyecto.

- Vivado ML Enterprise Edition: Una versión de pago que incluye soporte para todos los dispositivos de Xilinx, con opciones de licencia como Node Locked y Floating License, con precios que comienzan en $2,995 [10].

Estas herramientas y características hacen de Vivado una opción robusta y eficiente para diseñadores de hardware que trabajan con sistemas embebidos y FPGAs. Además, la plataforma ofrece una amplia gama de documentación, tutoriales y cursos de formación para ayudar a los desarrolladores a aprovechar al máximo sus capacidades.

2.3.1.3 Jupyter

Jupyter es una plataforma interactiva de desarrollo basada en la web, conocida principalmente por su interfaz de cuadernos interactivos. Es un proyecto de código abierto que promueve el software libre, los estándares abiertos y los servicios web para la informática interactiva en múltiples lenguajes de programación.

JupyterLab, la última interfaz de cuaderno de Jupyter, ofrece un entorno de desarrollo interactivo para cuadernos, código y datos. Su interfaz flexible permite a los usuarios configurar y organizar flujos de trabajo en ciencia de datos, computación científica, periodismo computacional y aprendizaje automático. Un diseño modular invita a extensiones que amplían y enriquecen su funcionalidad.

El Jupyter Notebook, por otro lado, es la interfaz original de aplicación web para crear y compartir documentos computacionales. Ofrece una experiencia sencilla y centrada en el documento. Los cuadernos de Jupyter soportan más de 40 lenguajes de programación, incluyendo Python, R, Julia y Scala. Los cuadernos pueden compartirse con otros a través de email, Dropbox, GitHub y el Jupyter Notebook Viewer.

Los cuadernos de Jupyter son conocidos por su capacidad para generar salidas interactivas que incluyen HTML, imágenes, vídeos, LaTeX[8] y tipos MIME[9] personalizados. También se integran con herramientas de big data como Apache Spark, y permiten la exploración de datos con herramientas como pandas, scikit-learn, ggplot2 y TensorFlow.

Jupyter también ofrece JupyterHub, una versión multiusuario del cuaderno diseñada para empresas, aulas y laboratorios de investigación. JupyterHub incluye autenticación enchufable y permite una implementación centralizada, siendo amigable con contenedores como Docker y Kubernetes.

Finalmente, Jupyter promueve estándares abiertos que permiten a los desarrolladores de terceros construir aplicaciones personalizadas. Los cuadernos de Jupyter son un formato de documento abierto basado en JSON que contiene un registro completo de la sesión del usuario, incluyendo código, texto narrativo, ecuaciones y salidas enriquecidas [11][12][13].

2.3.2 Proceso de diseño y simulación en FPGA/SoC para procesado de vídeo

El proceso de diseño y simulación en FPGA/SoC para procesado de vídeo utilizando las herramientas Vitis HLS, Vivado y Jupyter es un proceso integrado y multifacético que implica varios pasos clave, desde el desarrollo inicial hasta la implementación y las pruebas finales.

Desarrollo de filtros en Vitis HLS

1. Diseño de Alto Nivel: Comienza con la definición de los filtros de procesado de vídeo en un lenguaje de alto nivel como C/C++. Este enfoque permite una iteración rápida y una fácil comprensión del algoritmo. Es necesario aclarar que en realidad, C y C++ se consideran lenguajes de programación de "nivel medio" o "nivel intermedio", ya que ofrecen un equilibrio entre la abstracción del hardware y la portabilidad del software. Estos lenguajes permiten un control más directo sobre el hardware en comparación con los lenguajes de alto nivel como Python o Java, pero aún ofrecen cierto nivel de abstracción sobre los detalles específicos del hardware, lo que los hace más fáciles de

[8] LaTeX: Es un sistema de composición de documentos utilizado para producir documentos de alta calidad tipográfica. Utiliza un lenguaje de marcado para estructurar y formatear documentos, permitiendo a los usuarios concentrarse en el contenido mientras LaTeX se encarga del diseño y la presentación. Es especialmente popular para la creación de artículos científicos, tesis, informes técnicos y libros, gracias a su capacidad para gestionar fórmulas matemáticas, gráficos, referencias bibliográficas y otros elementos complejos de manera eficiente.

[9] MIME: Multipurpose Internet Mail Extensions, es un estándar de Internet que define los formatos para la transmisión de correos electrónicos y otros tipos de datos multimedia a través de la red. Permite que los mensajes de correo electrónico contengan no solo texto sin formato, sino también imágenes, audio, vídeo y otros tipos de archivos adjuntos. Además, especifica cómo se codifican y decodifican estos diferentes tipos de contenido para que puedan ser intercambiados entre diferentes sistemas de correo electrónico de manera efectiva y segura.

usar que los lenguajes de bajo nivel como el lenguaje ensamblador. En el contexto de la High-Level Synthesis (HLS), se consideran lenguajes de alto nivel aquellos que están por encima del nivel de abstracción del hardware, lo que incluye a C y C++, ya que permiten a los diseñadores expresar algoritmos y funcionalidades sin preocuparse por los detalles específicos de la arquitectura del hardware. Sin embargo, es importante tener en cuenta que, en comparación con lenguajes como Python o Java, C y C++ están más cerca del hardware y ofrecen un mayor control sobre el rendimiento y los recursos del sistema.

2. Síntesis de Alto Nivel: Utilizando Vitis HLS, este código de alto nivel se convierte en una descripción de hardware (a nivel RT). Durante este proceso, se puede aplicar directivas para optimizar el diseño para rendimiento, área, y consumo de energía. La High-Level Synthesis (HLS), o síntesis de alto nivel, es una técnica de diseño de hardware digital que permite la transformación automática de descripciones de alto nivel, como código en lenguajes de programación de alto nivel (por ejemplo, C, C++, SystemC), en circuitos digitales de hardware. Esta técnica es utilizada principalmente en el diseño de sistemas embebidos y de hardware de alto rendimiento.

3. Validación y Simulación: Antes de integrar el filtro en el diseño más grande, es esencial simular y validar su funcionamiento en Vitis HLS. Esto garantiza que el comportamiento del hardware coincida con el del algoritmo original.

Integración en Vivado

El proceso de integración en Vivado implica varios pasos clave detallados a continuación:

1. Importación de IP y Diseño del Sistema: La primera etapa involucra la importación de la IP[10] generada en Vitis HLS al entorno de diseño de Vivado. Esta IP puede incluir módulos de hardware desarrollados en C, C++ u otro lenguaje de alto nivel en Vitis HLS. Una vez importada, esta IP se integra con otros componentes necesarios del sistema en el entorno de diseño de Vivado. Estos componentes pueden incluir interfaces de memoria, lógica de control, módulos de conectividad y otros bloques de diseño necesarios para la funcionalidad del sistema.

2. Síntesis y Place & Route: En esta etapa, el diseño completo, que ahora incluye la IP importada de Vitis HLS y otros componentes del sistema, se somete a una síntesis adicional y al proceso de place & route en Vivado.

[10] IP: Intellectual Property se refiere a módulos o bloques de diseño predefinidos y reutilizables que se pueden integrar en proyectos de hardware digital. Estos módulos están diseñados para realizar funciones específicas, como procesamiento de señales, comunicación, interfaz con periféricos, entre otros. Permite a los diseñadores ahorrar tiempo y esfuerzo al utilizar componentes predefinidos y probados en lugar de diseñarlos desde cero. Esto facilita la creación de diseños complejos y acelera el proceso de desarrollo de hardware.

Durante la síntesis, el código del diseño se traduce en una estructura lógica que se puede implementar en el hardware del FPGA/SoC. Este proceso optimiza el diseño para cumplir con los requisitos de rendimiento, área y consumo de energía. Luego, el proceso de place & route coloca los elementos del diseño en la matriz de recursos del FPGA/SoC y realiza las conexiones entre ellos. Esto implica determinar la ubicación física de los elementos de diseño y trazar las rutas de conexión entre ellos.

3. Generación de Bitstream: Una vez completada la síntesis y el place & route, se genera un archivo bitstream. Este archivo contiene la información necesaria para programar el FPGA/SoC con el diseño desarrollado. El bitstream representa la configuración específica del hardware que implementa la funcionalidad deseada. Es un archivo binario que contiene la configuración de los bloques lógicos, las rutas de conexión y otros parámetros necesarios para definir el comportamiento del hardware. El bitstream generado se utiliza para programar el FPGA/SoC, lo que carga el diseño en el dispositivo y lo hace operativo.

Implementación y pruebas en Jupyter

1. Carga del Bitstream en la Pynq-Z2: Utilizando Jupyter, el bitstream generado en Vivado se carga en la placa Pynq-Z2. Esto prepara la placa para ejecutar el diseño de hardware.

2. Desarrollo de Código de Prueba: En Jupyter, se escribe código para interactuar con el FPGA/SoC y ejecutar el procesado de vídeo. Este código controla la carga del bitstream, maneja la entrada/salida de datos y ejecuta el procesado.

3. Ejecución y Comparación: El código desarrollado en Jupyter permite la ejecución del procesado de vídeo tanto en hardware (usando el FPGA/SoC) como en software (ejecución en la CPU). Esto facilita una comparación directa en términos de rendimiento, eficiencia y calidad del procesado.

4. Análisis de Resultados: Los resultados obtenidos de las pruebas de hardware y software se analizan para evaluar la eficacia del diseño en términos de velocidad, eficiencia energética y precisión del procesado de vídeo.

Este flujo de trabajo integral asegura un desarrollo eficiente y efectivo de sistemas de procesado de vídeo en hardware, aprovechando las ventajas de cada herramienta en cada etapa del proceso.

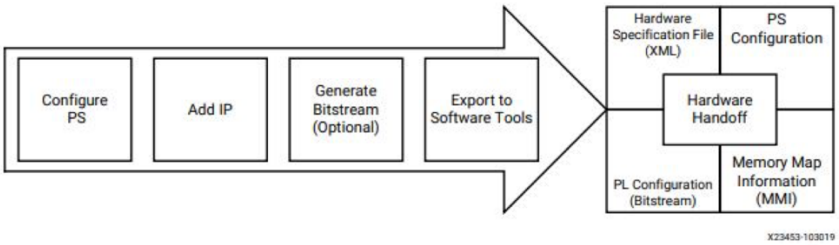

Figura 5 - Flujo de diseño. Imagen extraída de [14]

2.3.3 Plataforma Pynq-Z2

La Plataforma Pynq-Z2 es una placa diseñada para facilitar el desarrollo y la implementación de sistemas embebidos de alto rendimiento, especialmente en aplicaciones que requieren procesamiento intensivo de datos como el procesado de vídeo. Esta sección se enfocará en el SoC Zynq 7020, que se encuentra en el corazón de la plataforma Pynq-Z2, detallando sus especificaciones técnicas, la arquitectura ARM que soporta y las ventajas para el procesado de vídeo.

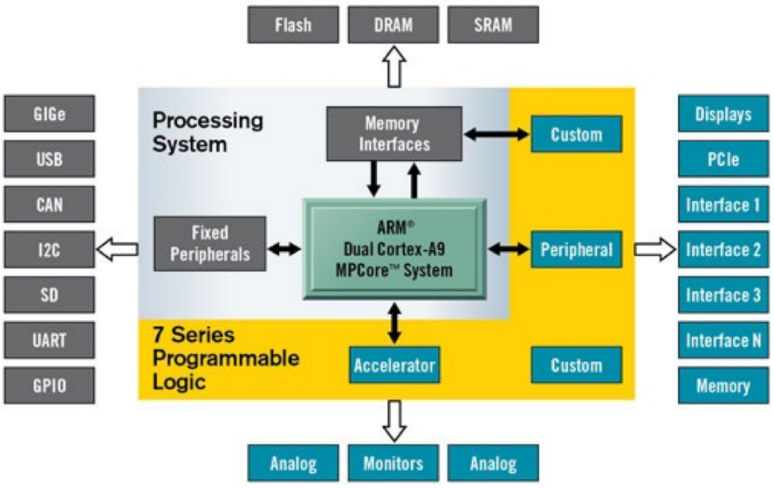

Figura 6 – Diagrama de bloques de los overlays de PYNQ. Imagen extraída de [15]

2.3.3.1 Especificaciones técnicas

Los elementos más significativos que contiene la Pynq-Z2 son:

- FPGA: Zynq-7000 SoC XC7Z020-1CLG400C
- Interfaces de E/S:
    - Conexión USB-JTAG programable electrónicamente

  - USB OTG 2.0
  - USB-UART bridge
  - 1 puerto 10/100/1G Ethernet
  - Entrada HDMI
  - Salida HDMI
  - Interfaz I2S con DAC de 24 bits con conector TRRS de 3,5 mm
  - Línea de entrada con conector de 3,5 mm

- Memoria:
  - 512 Mbyte DDR3 con bus de 16 bits a 1050 Mbps
  - 128 Mbit Quad-SPI Flash
  - Conector de tarjeta Micro SD

- Switches y LEDs:
  - 2 interruptores deslizantes
  - 2 LED RGB
  - 4 LED
  - 4 pulsadores

- Relojes:
  - 1 de 125 MHz para PL
  - 1 de 50 MHz para PS

- Puertos de expansión:
  - 2 puertos Pmod

  - 16 E/S FPGA total (8 pines compartidos con conector Raspberry Pi)
  - 1 conector de escudo Arduino
  - 24 E/S FPGA total
  - 6 entradas analógicas de un solo extremo de 0-3.3V a XADC[11]
  - Conector Raspberry Pi
  - 28 E/S FPGA total (8 pines compartidos con puerto Pmod A)

- Monitoreo de energía: Supervisión activa de las corrientes y voltajes de la fuente de alimentación.

- Hardware: Zynq-7000 SoC
  - Dual ARM® Cortex™-A9 MPCore™ con CoreSight™
  - 32 KB de instrucción, 32 KB de datos por procesador L1 Cache
  - Caché L2 unificado de 512 KB
  - 256 KB de memoria en chip
  - 2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO
  - 2x USB 2.0 (OTG), 2x Gigabit Ethernet de tres modos, 2 periféricos SD/SDIO en el chip
  - 85K celdas lógicas (13300 logic slices, cada una con cuatro LUTs de 6 entradas y 8 flip-flops)
  - 630 KB de fast block RAM

[11] XADC: Significa “Analog-to-Digital Converter” (Convertidor Analógico a Digital) integrado en los dispositivos FPGA de la serie Xilinx. La “X” en XADC proviene de Xilinx, el fabricante de estos dispositivos.

- Cuatro fichas de gestión del reloj, cada una con bucle de bloqueo de fase (PLL)
- 220 slices de DSP
- Velocidades de reloj internas superiores a 450 MHz
- 2x 12 bits, 1 MSPS Convertidor analógico-digital en el chip (XADC)

Información obtenida de [16].

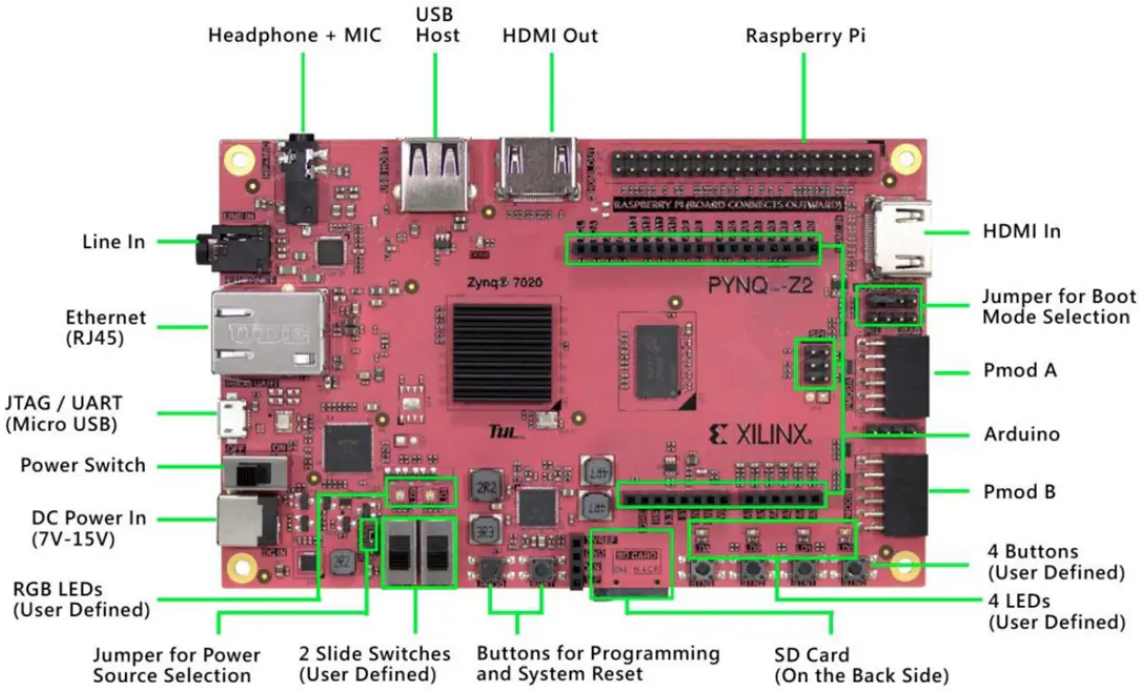

Figura 7 - Imagen de una PYNQ-Z2 y sus componentes. Imagen extraída de [17]

Una característica distintiva de la plataforma PYNQ-Z2, es la implementación y uso de **overlays**. Los overlays son diseños de FPGA descargables que actúan como capas de configuración sobre la lógica programable del dispositivo, permitiendo la reconfiguración dinámica del hardware para realizar diferentes tareas o funciones específicas sin necesidad de modificar el diseño físico del chip.

En la PYNQ-Z2, los overlays juegan un papel crucial al permitir que los desarrolladores aprovechen la flexibilidad de la lógica programable (PL) del Zynq-7000 SoC para aplicaciones específicas, mientras mantienen el sistema de procesamiento (PS), compuesto por los núcleos ARM Cortex-A9, dedicado a la gestión general del sistema y tareas de control.

Los overlays permiten que ciertos periféricos y funciones, tales como la entrada/salida HDMI, interfaces de comunicación como Ethernet y USB, así como la conectividad con otros módulos como PMOD y Arduino, sean mapeados dinámicamente hacia la lógica programable. Esto significa que dependiendo del overlay cargado, la PL puede ser configurada para manejar específicamente tareas de procesamiento de imagen y vídeo, liberando así recursos del PS para otras operaciones.

La arquitectura de la PYNQ-Z2 está diseñada de tal manera que los overlays no solo facilitan la interacción entre el PS y la PL, sino que también permiten una integración fluida y eficiente de hardware y software. Al utilizar Jupyter Notebooks, por ejemplo, los desarrolladores pueden cargar fácilmente overlays en la placa, experimentar con diferentes configuraciones de hardware y desarrollar aplicaciones de procesamiento de vídeo con una complejidad y eficiencia que sería difícil de lograr con arquitecturas de hardware fijas.

Este enfoque de overlays subraya la filosofía de diseño de la PYNQ-Z2: proporcionar una plataforma versátil y accesible para la innovación en aplicaciones, donde la reconfiguración en tiempo real y la optimización del rendimiento son fundamentales.

2.3.3.2 Características del Zynq 7020

Especificaciones técnicas y ventajas para procesado de vídeo

El Zynq-7020 combina un alto rendimiento de procesamiento de aplicaciones con funcionalidades específicas de FPGA (Field Programmable Gate Array), lo que lo hace ideal para el procesado de vídeo. A continuación, se detallan sus especificaciones técnicas y ventajas:

- Especificaciones Técnicas:
    - CPU: Dual-core ARM Cortex-A9 MPCore con una frecuencia de hasta 866 MHz.
    - FPGA: Artix-7 clase FPGA con aproximadamente 85,000 celdas lógicas, lo que permite la implementación de lógica personalizada y algoritmos de procesamiento de vídeo de alta velocidad.
    - Memoria: Hasta 1 GB de DDR3 con un ancho de banda de 32 bits y velocidades de hasta 1050 Mbps, esencial para el manejo de grandes volúmenes de datos en aplicaciones de vídeo.
    - Periféricos: Incluye HDMI, USB, Ethernet, y otras interfaces que facilitan la captura, salida, y comunicación de datos de vídeo.

- Ventajas para el Procesado de Vídeo:

    - Flexibilidad: La combinación de procesadores ARM y lógica programable FPGA permite a los desarrolladores optimizar algoritmos de procesado de vídeo, balanceando entre eficiencia energética y rendimiento computacional.

    - Alto Rendimiento: Capacidad para implementar filtros de vídeo y algoritmos de procesamiento en tiempo real gracias a su potente FPGA.

    - Integración: Facilidad para conectar con sensores y cámaras a través de sus múltiples interfaces, permitiendo una captura de vídeo eficiente y directa.

Arquitectura ARM y su rol en la gestión de procesos

La arquitectura ARM del Zynq-7020 desempeña un papel crucial en la gestión de procesos, especialmente en aplicaciones de procesado de vídeo, debido a sus características:

- Gestión Eficiente: Los dos núcleos ARM Cortex-A9 permiten una gestión eficiente del sistema operativo y las aplicaciones de usuario, distribuyendo la carga de trabajo de manera efectiva.

- Soporte de Software: Amplio soporte de herramientas de desarrollo y sistemas operativos, facilitando la implementación de algoritmos de procesado de vídeo y la interacción con la lógica FPGA.

- Interoperabilidad: Los procesadores ARM pueden funcionar en conjunto con la lógica FPGA para preprocesar datos de vídeo antes de su procesamiento intensivo en la FPGA, o para realizar tareas de control y comunicaciones.

Referencias:

Las especificaciones y ventajas discutidas se basan en la documentación técnica del Zynq-7020 y la plataforma Pynq-Z2 proporcionada por Xilinx, así como en principios generales de diseño de sistemas embebidos y procesado de vídeo. Para información más detallada, se recomienda consultar:

- Xilinx. "Zynq-7000 SoC Data Sheet: Overview." [18]

- PYNQ. "PYNQ-Z2: Python Productivity for Zynq." [19]

2.3.3.3 Herramientas utilizadas: Vitis HLS, Vivado y Jupyter

El desarrollo de aplicaciones de procesado de vídeo en hardware, especialmente en plataformas como Pynq-Z2, implica el uso de varias herramientas especializadas. Vitis HLS, Vivado, y Jupyter constituyen una cadena de herramientas integral que facilita desde la concepción del diseño hasta su implementación y evaluación.

Integración y flujo de trabajo entre las herramientas

El flujo de trabajo entre Vitis HLS, Vivado, y Jupyter se organiza en etapas secuenciales, cada una con un propósito específico en el desarrollo de aplicaciones de procesado de vídeo:

1. Diseño y Simulación con Vitis HLS: Desarrollo de la lógica de procesamiento en un alto nivel de abstracción.

2. Integración y Síntesis con Vivado: Integrar la IP generada por Vitis HLS en un diseño de sistema más amplio y preparar el diseño para la implementación en hardware.

3. Evaluación y Demostración con Jupyter: Facilitar la interacción con el diseño de hardware implementado y realizar comparaciones de rendimiento entre las versiones de hardware y software del algoritmo de procesado de vídeo.

Información obtenida de [20].

2.3.3.4 Configuración de la placa: SD, Linux, etc.

La configuración inicial de la placa Pynq-Z2 es un paso crucial para asegurar que la plataforma esté preparada para el desarrollo y ejecución de aplicaciones de procesado de vídeo. Este proceso implica la preparación de la tarjeta SD con una imagen del sistema operativo, la configuración del sistema Linux que se ejecuta en la placa, y el cumplimiento de los requisitos del sistema necesarios para un funcionamiento óptimo. A continuación, se explica brevemente el proceso, para más detalle, en el Anexo tutorial se puede encontrar el proceso de preparación de la tarjeta SD paso a paso.

Configuración inicial y requisitos del sistema

La configuración inicial de la placa Pynq-Z2 se centra en varios pasos esenciales que preparan el dispositivo para el desarrollo:

1. Preparación de la Tarjeta SD: Tarjeta SD de al menos 8 GB de capacidad y clase 10 para un rendimiento óptimo. La imagen oficial del sistema operativo PYNQ (basado en Linux) se puede descargar desde el sitio web de PYNQ: http://www.pynq.io/board.html. La imagen debe ser grabada en la tarjeta SD

utilizando herramientas como Etcher, Win32 Disk Imager, o dd en sistemas Linux.

2. Arranque desde la Tarjeta SD: Insertar la tarjeta SD en la ranura de la placa Pynq-Z2. Conectar la placa a una fuente de alimentación y a una red mediante un cable Ethernet o configuración Wi-Fi (si se soporta). La placa debería arrancar automáticamente desde la tarjeta SD.

3. Acceso al Sistema:

- Vía Ethernet: A través de la red, utilizando “ssh” para acceder a la terminal del sistema operativo Linux en la placa. La dirección IP puede obtenerse mediante el escaneo de la red o utilizando un monitor y un teclado conectados directamente a la placa.

- Vía Jupyter Notebook: Accediendo a la interfaz web de Jupyter Notebook a través de un navegador, ingresando la dirección IP de la placa seguido de “:9090” (por ejemplo, “http://192.168.2.99:9090”).

4. Requisitos del Sistema y Configuraciones Adicionales: Es recomendable actualizar el sistema operativo y los paquetes a las últimas versiones disponibles usando el gestor de paquetes “apt”. Si es necesario, configurar la red Wi-Fi (en caso de que se use un adaptador Wi-Fi USB compatible) a través de la interfaz de línea de comandos o herramientas de configuración de red.

5. Instalación de Herramientas y Bibliotecas Específicas:

- Instalación de herramientas necesarias para el desarrollo, como compiladores, editores de texto, y herramientas específicas de Xilinx (por ejemplo, Vitis HLS y Vivado) si se requieren para la preparación de la aplicación antes de la carga en la placa.

- Bibliotecas de Procesado de Vídeo: Instalación de bibliotecas específicas para el procesado de vídeo que se utilizarán en el proyecto, tales como OpenCV para el procesamiento de imágenes y vídeo en el lado del software.

Información obtenida de [21].

La configuración inicial y los requisitos del sistema aseguran que la placa Pynq-Z2 esté lista para el desarrollo y la ejecución eficiente de aplicaciones de procesado de vídeo, proporcionando una plataforma robusta para la exploración y experimentación en el campo del procesado de vídeo en hardware.

Interacción entre el hardware de la placa y el software

La interacción efectiva entre el hardware de la placa Pynq-Z2 y el software es fundamental para el éxito de proyectos de procesado de vídeo en hardware. Esta sinergia permite aprovechar al máximo las capacidades de la placa, facilitando la implementación de soluciones de procesado de vídeo eficientes y de alto rendimiento.

Aspectos clave de la interacción hardware-software

1. Acceso a Recursos de Hardware: El sistema operativo Linux y las herramientas de desarrollo proporcionan mecanismos para acceder y controlar los recursos de hardware de la placa, como la FPGA, los procesadores ARM, y los periféricos. Esto se logra a través de interfaces de programación de aplicaciones (APIs), drivers específicos del dispositivo, y herramientas de configuración de bajo nivel.

2. Uso de la FPGA: La FPGA en el Zynq-7020 permite una personalización profunda del hardware para tareas específicas de procesado de vídeo. Los desarrolladores pueden diseñar lógica de hardware personalizada usando Vitis HLS y Vivado para implementar filtros de vídeo, algoritmos de compresión, o cualquier otra función de procesamiento de vídeo, cargando luego el diseño a la FPGA a través del bitstream generado.

3. Programación y Desarrollo: Para la implementación de algoritmos de procesado de vídeo y la interacción con el hardware, se pueden utilizar tanto C/C++ (para un control más directo del hardware y mayor eficiencia) como Python (para una mayor facilidad de desarrollo y experimentación). El uso de APIs y bibliotecas como OpenCV en Python facilita la implementación de complejos algoritmos de procesado de vídeo sin necesidad de gestionar directamente el hardware.

4. Comunicación entre Procesadores ARM y la FPGA: La arquitectura de la placa permite una comunicación fluida entre los procesadores ARM y la FPGA, lo que es esencial para el procesado de vídeo en tiempo real. Los datos de vídeo pueden ser capturados y preprocesados por el software que se ejecuta en los procesadores ARM antes de ser enviados a la FPGA para su procesamiento intensivo.

5. Interfaz de Usuario y Visualización: Como, por ejemplo, Jupyter Notebooks, que proporciona una interfaz interactiva para el desarrollo y demostración de aplicaciones de procesado de vídeo. Los desarrolladores pueden escribir código, ejecutarlo, y visualizar los resultados en tiempo real, facilitando la experimentación y el ajuste fino de algoritmos.

Consideraciones prácticas

Para aplicaciones de procesado de vídeo en tiempo real, es crucial optimizar tanto el software (algoritmos, uso de bibliotecas, etc.) como el diseño de hardware (utilización de la FPGA) para alcanzar los requisitos de rendimiento.

La plataforma Pynq-Z2 facilita la depuración y el testeo de aplicaciones de procesado de vídeo, permitiendo iteraciones rápidas entre el desarrollo de software y hardware [22][23].

La interacción entre el hardware de la placa y el software es un componente crítico en el desarrollo de aplicaciones de procesado de vídeo en hardware, permitiendo a los desarrolladores explotar las capacidades únicas de la placa Pynq-Z2 para implementar soluciones innovadoras y eficientes.

3 PRUEBAS PRELIMINARES

El desarrollo de sistemas embebidos para procesamiento de vídeo en hardware requiere una comprensión detallada de las herramientas y tecnologías involucradas. Este capítulo documenta las pruebas preliminares realizadas para establecer un entorno de desarrollo robusto, probar la comunicación básica con hardware (PYNQ-Z2) y validar el diseño inicial en Vitis HLS.

3.1 Pruebas iniciales y configuración

La fase de pruebas iniciales y configuración es esencial para asegurar que el entorno de desarrollo y las herramientas estén correctamente establecidos para el desarrollo eficiente de aplicaciones de procesado de vídeo en hardware. Esta fase ayuda a identificar y resolver problemas temprano en el ciclo de desarrollo, asegurando que el proyecto se base en un fundamento sólido y confiable.

3.1.1 Configuración del entorno de desarrollo

El entorno de desarrollo se basa en una combinación de herramientas que incluyen Vitis HLS para diseño de hardware de alto nivel, Vivado para la integración del sistema y diseño FPGA, y Jupyter Notebooks para la interacción con el SoC PYNQ-Z2. La elección de estas herramientas se debe a su capacidad para ofrecer un flujo de trabajo integrado desde el diseño hasta la implementación, permitiendo un rápido desarrollo y prueba de prototipos.

Para comenzar, se hicieron unos diseños sencillos que ayudaron a poner en pie el flujo de diseño y garantizar que los diseños se estaban realizando correctamente. Estas pruebas han permitido crear un tutorial muy detallado de los pasos a seguir para que pueda seguirse por cualquier persona que quiera familiarizarse con el entorno PYNQ-Z2. Este tutorial puede encontrarse en el anexo correspondiente de esta memoria y consta de las siguientes partes:

1. Anexo Tutorial, Parte 1. Interacción con la plataforma Jupyter: Proporciona una introducción práctica a la interacción básica con el hardware usando Python en Jupyter Notebooks.

2. Anexo Tutorial, Parte 2. Control de LEDs en Vivado: Introduce la personalización y el desarrollo de hardware de bajo nivel, esencial para entender la creación de IP personalizada.

3. Anexo Tutorial, Parte 3. Configuración del reloj de la FPGA: Explora conceptos avanzados, incluyendo el trabajo con el reloj FPGA, crucial para el procesamiento de vídeo en tiempo real.

4. Anexo Tutorial, Parte 4. Primer diseño en Vitis HLS: Aporta los conocimientos básicos para desarrollar una IP en Vitis HLS y conocer el flujo de trabajo de

la placa desde su nivel más básico hasta su implementación y uso en Jupyter, pasando por Vivado.

3.2 Selección y justificación de las pruebas iniciales realizadas

La estrategia detrás de la selección de pruebas iniciales se basó en construir gradualmente una comprensión y habilidad en el uso del ecosistema PYNQ-Z2, comenzando con tutoriales básicos y avanzando hacia aplicaciones de procesamiento de vídeo más complejas. Esta progresión aseguró no solo una base sólida en el manejo del hardware y software, sino también una comprensión práctica de cómo implementar y optimizar procesos de vídeo en el sistema.

3.2.1 Familiarización con PYNQ-Z2

La elección de tutoriales específicos fue el primer paso crítico, proporcionando conocimientos esenciales sobre la programación y configuración del PYNQ-Z2. Estos tutoriales aseguraron que se adquirieran las competencias básicas necesarias para desarrollar aplicaciones más complejas. La familiarización con la interacción de bajo nivel (LEDs, switches) y la comprensión de la síntesis de hardware (uso de Verilog/Vivado) son fundamentales para aprovechar al máximo las capacidades de la plataforma.

A continuación, se muestran los diseños de forma muy superficial, para más detalle, leer el Anexo Tutorial.

3.2.1.1 Interacción básica con la plataforma Jupyter

En esta sección nos enfocamos en establecer una sólida base para el manejo y aprovechamiento de la plataforma Jupyter Notebook, un elemento crucial para la interacción efectiva con la parte de lógica programable (PL) de la placa PYNQ-Z2. Este conocimiento es imprescindible para una exploración inicial de la placa y para el desarrollo avanzado de aplicaciones de procesamiento de vídeo. Abordamos cómo los overlays permiten una comunicación fluida y directa con la PL, una característica distintiva que aprovechamos extensamente. Este segmento sirve como un punto de partida, detallado en el Tutorial Parte 1: Interacción básica con Jupyter, para familiarizarse con las dinámicas de trabajo en Jupyter Notebook, preparando el terreno para implementaciones más complejas y especializadas en capítulos siguientes.

El código permite controlar el estado (encendido o apagado) de cuatro LEDs en la placa FPGA, basándose en la posición de dos switches físicos. Cada switch controla el estado de dos LEDs específicos: `sw0` controla `led0` y `led1`, mientras que `sw1` controla `led2` y `led3`. Es un ejemplo básico de cómo interactuar con hardware en tiempo real mediante programación de alto nivel.

Código del archivo TogglingLEDs.ipynb:

```
from time import sleep
from pynq.overlays.base import BaseOverlay
base = BaseOverlay("base.bit")

led0 = base.leds[0]     # Se asigna al LED LD0
led1 = base.leds[1]     # Se asigna al LED LD1
led2 = base.leds[2]     # Se asigna al LED LD2
led3 = base.leds[3]     # Se asigna al LED LD3

sw0 = base.switches[0]     # Se asigna al Switch SW0
sw1 = base.switches[1]     # Se asigna al Switch SW1

while(True):     # Se ejecuta siempre que while (True), es decir, siempre
    if(sw0.read() == True):     # Lee SW0 y comprueba si está ON
        led0.on()               # Si SW0 está ON --> Encender LED0
        led1.on()               # Si SW0 está ON --> Encender LED1
    else:
        led0.off()              # Sino, apagar LED0
        led1.off()              # Sino, apagar LED1

    if(sw1.read() == True):     # Lee SW1 y comprueba si está ON
        led2.on()               # Si SW1 está ON --> Encender LED2
        led3.on()               # Si SW1 está ON --> Encender LED3
    else:
        led2.off()              # Sino, apagar LED2

        led3.off()              # Sino, apagar LED3
```

1. Importaciones:

- `from time import sleep`: Importa la función `sleep` del módulo `time`, la cual puede usarse para hacer pausas durante la ejecución del código, aunque en el fragmento proporcionado no se utiliza, pero sí en los siguientes diseños.

- `from pynq.overlays.base import BaseOverlay`: Importa la clase `BaseOverlay` del módulo `pynq.overlays.base`. `BaseOverlay` se utiliza para cargar un archivo de diseño de circuito (.bit) en la FPGA, que configura los pines y otros recursos de la placa para interactuar con componentes específicos de la placa.

2. Configuración inicial:

- `base = BaseOverlay("base.bit")`: Carga el archivo de diseño `base.bit` en la FPGA. Este diseño incluye la configuración para interactuar con los elementos básicos, como los LEDs y switches.

- Se asignan los primeros cuatro LEDs (`led0` a `led3`) y los dos primeros switches (`sw0` y `sw1`) a variables para facilitar su manejo en el código.

3. Bucle infinito:

- `while(True)`: Crea un bucle infinito que permite al código ejecutarse continuamente.

4. Control de LEDs mediante Switches:

- Dentro del bucle, el código comprueba continuamente el estado (ON o OFF) de los switches `sw0` y `sw1` utilizando el método `.read()`.

- Si `sw0` está en ON (`True`), los LEDs `led0` y `led1` se encienden. Si está en OFF (`False`), estos LEDs se apagan.

- De manera similar, si `sw1` está en ON, los LEDs `led2` y `led3` se encienden. Si está en OFF, se apagan.

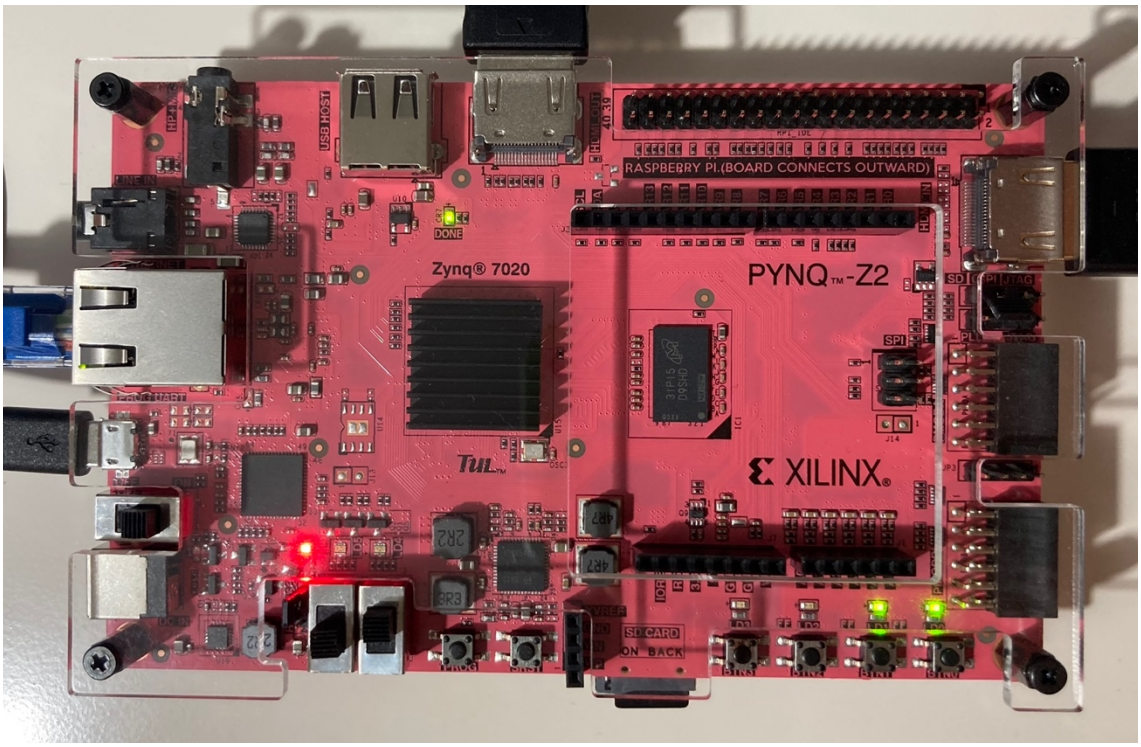

Figura 8 - Resultado de la prueba 1

Como puede comprobarse en la Figura 8, el Switch 1 está on, mientras el Switch 2 está off, como consecuencia del código desarrollado en Jupyter, los LEDs 0 y 1 permanecen encendidos y los LEDs 2 y 3 apagados.

3.2.1.2 Diseño complementario: control LEDs RGB

Dirigimos nuestra exploración en esta sección hacia una característica interactiva y visualmente atractiva de la placa PYNQ-Z2: los LEDs RGB. Este segmento es interesante para comprender cómo podemos utilizar estos elementos para añadir una dimensión visual a nuestros proyectos. Al dominar el control de los LEDs RGB, incorporamos una capa adicional de interacción y feedback en nuestros diseños, permitiéndonos comunicar el estado del sistema, alertas, o simplemente añadir un elemento estético. Este conocimiento, que se detalla en la primera parte del tutorial, es crucial para aquellos que buscan integrar señales visuales en sus proyectos, enriqueciendo la experiencia del usuario y facilitando la depuración y prueba de aplicaciones complejas.

El código es una demostración interactiva para la placa PYNQ, utilizando LEDs y botones. Inicialmente, enciende todos los LEDs. Mientras el botón 3 no se presione, se pueden realizar varias acciones: el botón 0 cambia los colores de los LEDs RGB, el botón 1 apaga todos los LEDs y luego los alterna entre encendido y apagado, y el botón 2 realiza la misma acción, pero en orden inverso. Al presionar el botón 3, se finaliza el programa y se apagan todos los LEDs.

Código del archivo RGBLEDs.ipynb:

```
from time import sleep     # Importa 'sleep' de biblioteca 'time' para pausas
from pynq.overlays.base import BaseOverlay     # Importa la clase 'BaseOverlay'

base = BaseOverlay("base.bit")    # Carga la configuración de la placa PYNQ

Delay1 = 0.3 # Define una pausa de 0.3 segundos
Delay2 = 0.1 # Define una pausa de 0.1 segundos
color = 0     # Inicializa una variable de color a 0
rgbled_position = [4, 5]   # Define posiciones LEDs RGB en la placa

for led in base.leds:      # Itera a través de los LEDs disponibles en la placa
    led.on()               # Enciende cada LED

while (base.buttons[3].read() == 0):    # Mientras botón 3 no presionado

    if (base.buttons[0].read() == 1):   # Si el botón 0 está presionado
        color = (color + 1) % 8  # Incrementa el valor 'color' de 0 a 7
        for led in rgbled_position:     # Itera a través de los LEDs RGB
            base.rgbleds[led].write(color)     # Valor 'color' en los LEDs RGB
            base.rgbleds[led].write(color)     # Valor 'color' en LEDs 4 y 5
        sleep(Delay1)                          # Pausa por 0.3 segundos

    elif (base.buttons[1].read() == 1):# Si el botón 1 está presionado
        for led in base.leds:    # Itera a través de todos los LEDs
            led.off()      # Apaga todos los LEDs
        sleep(Delay2)      # Pausa por 0.1 segundos
        for led in base.leds:    # Itera a través de todos los LEDs otra vez
            led.toggle()   # Cambia el estado de los LEDs (on/off)
            sleep(Delay2)  # Pausa por 0.1 segundos entre cambios de estado

    elif (base.buttons[2].read() == 1):# Si el botón 2 está presionado
        for led in reversed(base.leds):# Iterar LEDs en orden inverso
            led.off()      # Apagar todos los LEDs en orden inverso
        sleep(Delay2)      # Pausa por 0.1 segundos
        for led in reversed(base.leds):# Iterar LEDs en orden inverso
```

```
            led.toggle()   # Cambia el estado de los LEDs (on/off)
            sleep(Delay2)  # Pausa por 0.1 segundos entre cambios de estado

print('Fin de la prueba')  # Imprime un mensaje de finalización

for led in base.leds:      # Itera a través de todos los LEDs disponibles
    led.off()              # Apagar todos los LEDs
for led in rgbled_position:      # Itera a través de los LEDs RGB
    base.rgbleds[led].off()      # Apaga los LEDs RGB en las posiciones 4 y 5
```

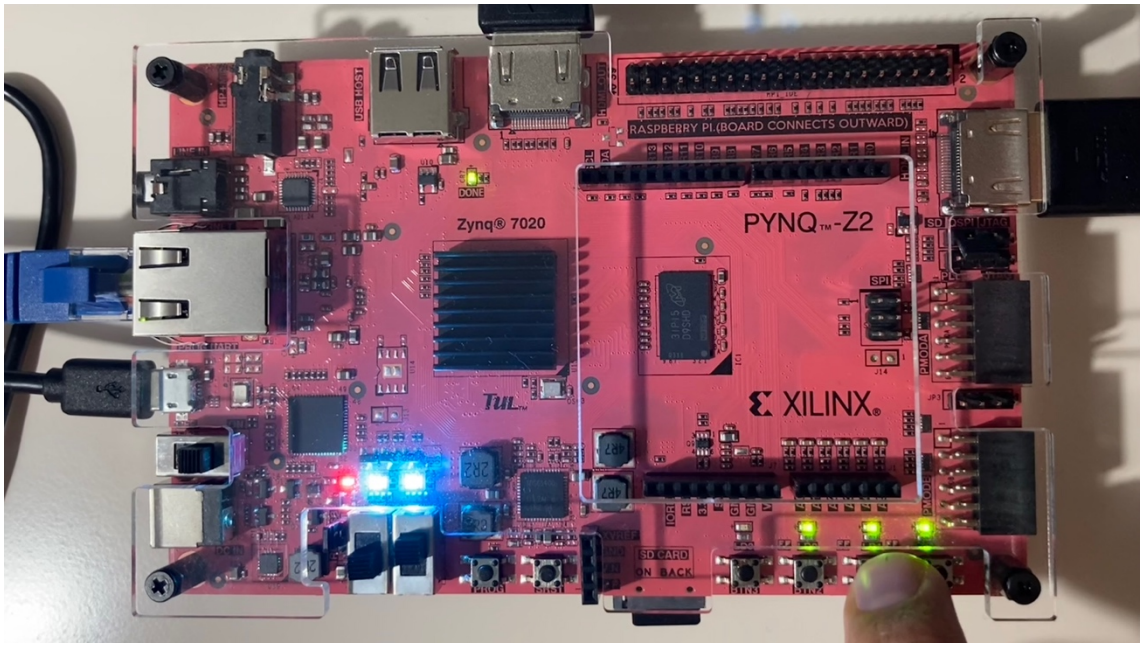

Figura 9 – Resultado de la prueba de los LEDs RGB

Como puede comprobarse en la Figura 9, al pulsar el botón 2 los LEDs del 0 al 3 se apagan y enciende siguiendo una secuencia, y los LEDs 4 y 5 cambian de color cuando se pulsa el botón 0.

3.2.1.3 Uso de diseños realizados en Vivado

Esta sección se enfoca en cómo utilizar Verilog y Vivado para programar LEDs en la placa PYNQ-Z2. Cubre desde la instalación de Vivado y la configuración de la placa, hasta la creación de un proyecto en Vivado, la escritura de código Verilog para controlar los LEDs, y la generación del bitstream para programar la FPGA. También incluye detalles sobre la creación de un módulo en Verilog, el diseño de bloques, la síntesis, y la implementación del proyecto en la placa.

Al integrar este aprendizaje, buscamos proporcionar una guía concreta y aplicable que enseñe el manejo técnico, y que también inspire a los usuarios a explorar aplicaciones innovadoras y creativas.

Código de la prueba 2:

Este código Verilog define un módulo llamado `BasicLEDController` que toma cuatro botones de entrada (`BTN`) y controla cuatro LEDs de salida (`LD`). La línea `assign LD = BTN;` significa que el estado de cada LED es directamente asignado al estado del botón correspondiente: cuando un botón se presiona (asumiendo que un '1' significa presionado), el LED correspondiente se enciende, y cuando el botón se suelta, el LED se apaga. Es un ejemplo simple de conexión directa entre entradas y salidas.

```
module BasicLEDController(
       input [3:0] BTN,
       output [3:0] LD
       );

       assign LD = BTN;
endmodule
```

Como resultado, obtenemos el diseño de la Figura 10, es un bloque IP que implementa el código Verilog especificado.

Figura 10 – Bloque diseñado en Verilog, BasicLEDController

El siguiente código define un módulo `MainDesign_wrapper` que sirve como un envoltorio o "wrapper" para otro módulo `MainDesign`. Los puertos de entrada (`BTN`) y salida (`LD`) se declaran tanto para `MainDesign_wrapper` como internamente con conexiones tipo `wire`. Luego, se instancia el módulo `MainDesign` dentro del `MainDesign_wrapper`, conectando directamente sus entradas y salidas a los del wrapper. Esto permite que `MainDesign_wrapper` funcione como una interfaz para `MainDesign`, facilitando su integración o modificación sin alterar el módulo original.

```
module MainDesign_wrapper
       (BTN,
       LD);
       input [3:0]BTN;
       output [3:0]LD;
       wire [3:0]BTN;
       wire [3:0]LD;
       MainDesign MainDesign_i
              (.BTN(BTN),
              .LD(LD));

Endmodule
```

El `MainDesign_wrapper` actúa como un contenedor o envoltorio para nuestro diseño principal, facilitando la comunicación entre la lógica programable (PL) y el sistema de procesamiento (PS) dentro del SoC. La generación de este wrapper puede realizarse de manera automática o manual, dependiendo de la herramienta de desarrollo utilizada, siendo comúnmente Vivado el entorno donde este proceso se lleva a cabo.

La necesidad de incluir el `MainDesign_wrapper` radica en su capacidad para encapsular el diseño específico de usuario, permitiendo una integración y sincronización eficaz con los componentes estándar del SoC, como los núcleos ARM y las interfaces de comunicación. En Vivado, la generación de este wrapper suele ser automática cuando se finaliza el diseño del bloque IP o al integrar diversos módulos IP en el IP Integrator, simplificando así el proceso de preparación para la síntesis y la implementación.

Una vez el `MainDesign_wrapper` está en su lugar, el siguiente paso es la asignación de pines, que se realiza en el entorno de Vivado mediante el archivo de restricciones de diseño (XDC). Este archivo define la conexión física entre los pines del FPGA y los diferentes periféricos y interfaces de la placa PYNQ-Z2, asegurando que la lógica diseñada se comunique correctamente con el mundo exterior.

Con el diseño sintetizado y los pines correctamente asignados, procedemos a la generación del archivo .bit. Este archivo es esencialmente una imagen binaria que contiene toda la configuración de la FPGA, lista para ser cargada en el dispositivo. Vivado facilita este proceso, ofreciendo una secuencia de pasos que lleva desde la síntesis del diseño hasta la generación del archivo .bit, pasando por las fases de implementación y verificación.

El archivo .bit generado es un componente fundamental del overlay de PYNQ, que puede ser un overlay existente que se está modificando o uno completamente nuevo que se está creando. El proceso de inclusión en el overlay implica integrar el archivo .bit junto con un archivo de descripción de hardware (usualmente en formato HWH o TCL) en el entorno de desarrollo de Python de PYNQ. Esta combinación permite a los usuarios de PYNQ cargar dinámicamente el diseño FPGA en la placa e interactuar con él mediante scripts de Python, aprovechando las bibliotecas PYNQ para facilitar la comunicación entre el PS y el PL.

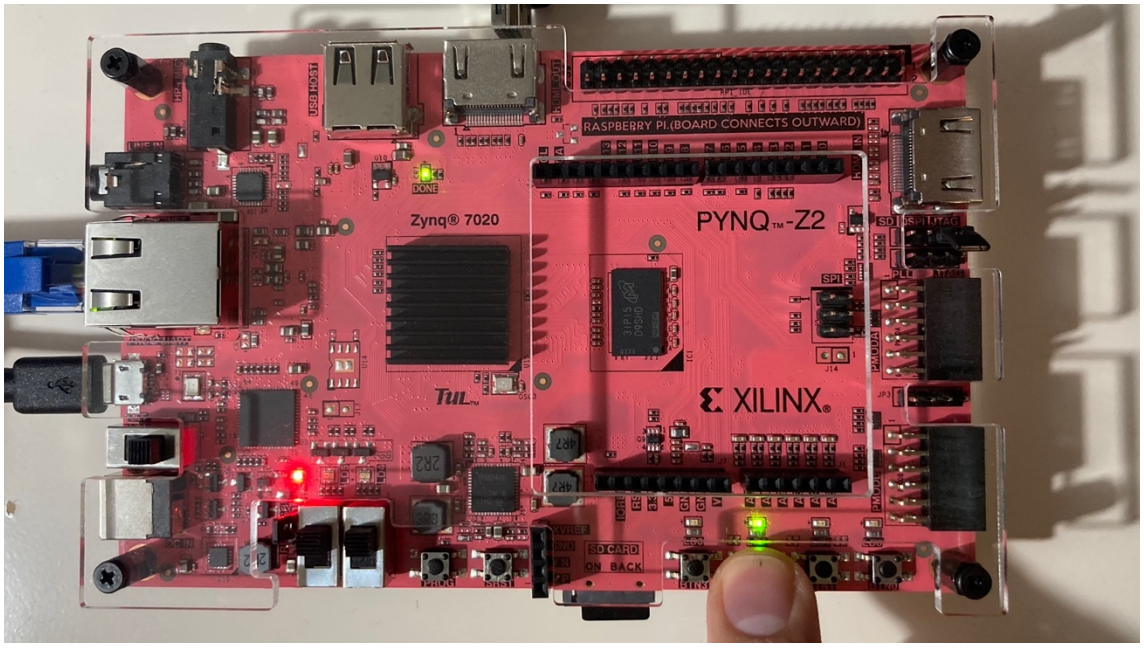

Figura 11 – Placa programada con el diseño de la parte 2 del Tutorial

Observando la Figura 11, se puede comprobar cómo los botones están directamente conectados a los LEDs, en este caso el botón 2 al LED 2.

3.2.1.4 Modificación de la entrada de reloj

La parte 3 del tutorial enseña cómo usar la fuente de reloj de 125 MHz disponible en la placa PYNQ-Z2 para hacer parpadear LEDs a diferentes frecuencias mediante divisores de frecuencia de reloj. Se explica cómo diseñar y sintetizar un divisor de reloj en Verilog con Vivado, que divide la señal de reloj de entrada a frecuencias más lentas. Luego, estas señales de reloj divididas se utilizan para controlar el parpadeo de LEDs a velocidades variables, demostrando el uso práctico de relojes y divisores en el diseño de circuitos FPGA.

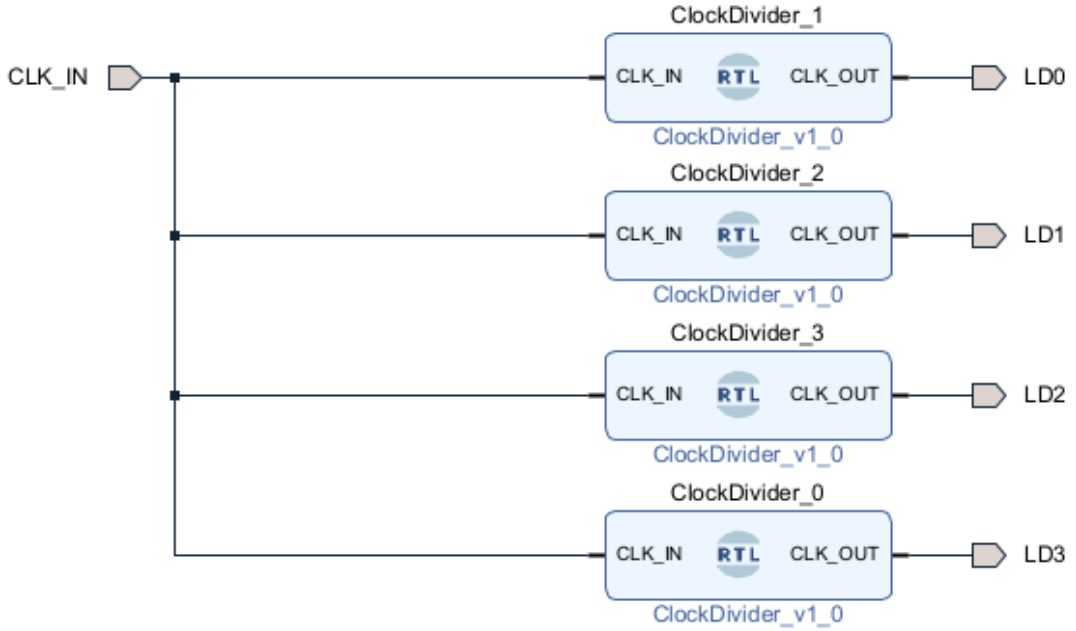

Figura 12 – Diagrama de bloques del diseño, 4 bloques ClockDivider a distintas frecuencias

Vivado también nos permite obtener el diseño del diagrama de bloques a más bajo nivel, como un conjunto de puertas lógicas, como se puede observar en la Figura 13.

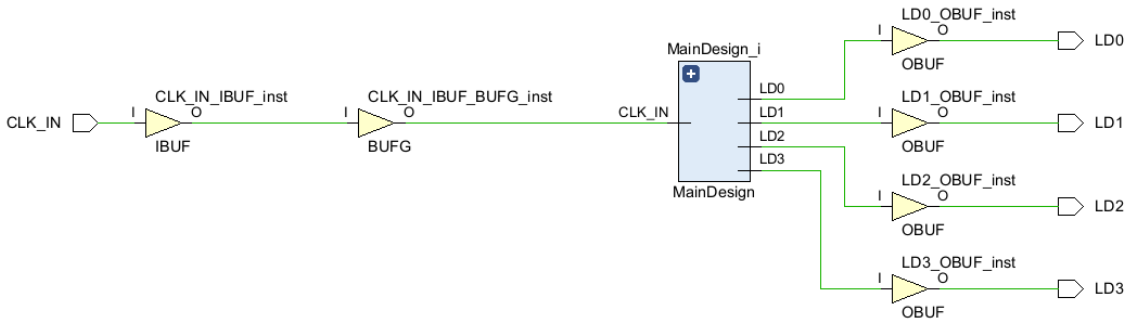

Figura 13 – Diseño a bajo nivel del diagrama de la Figura 12

En la figura 14 se muestra el análisis de potencia a partir de la lista de conjuntos implementada. Actividad derivada de archivos de restricciones, archivos de simulación o análisis vectorless.

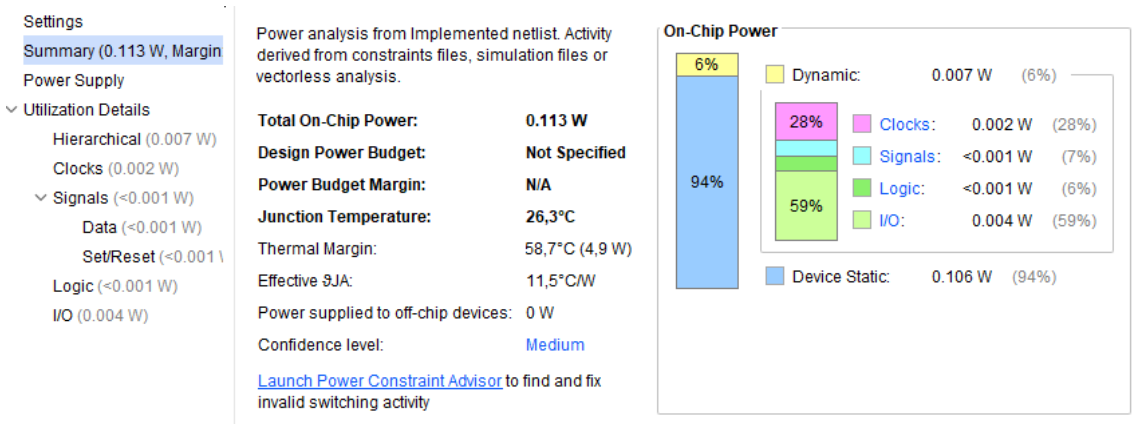

Figura 14 – Resumen del análisis de potencia

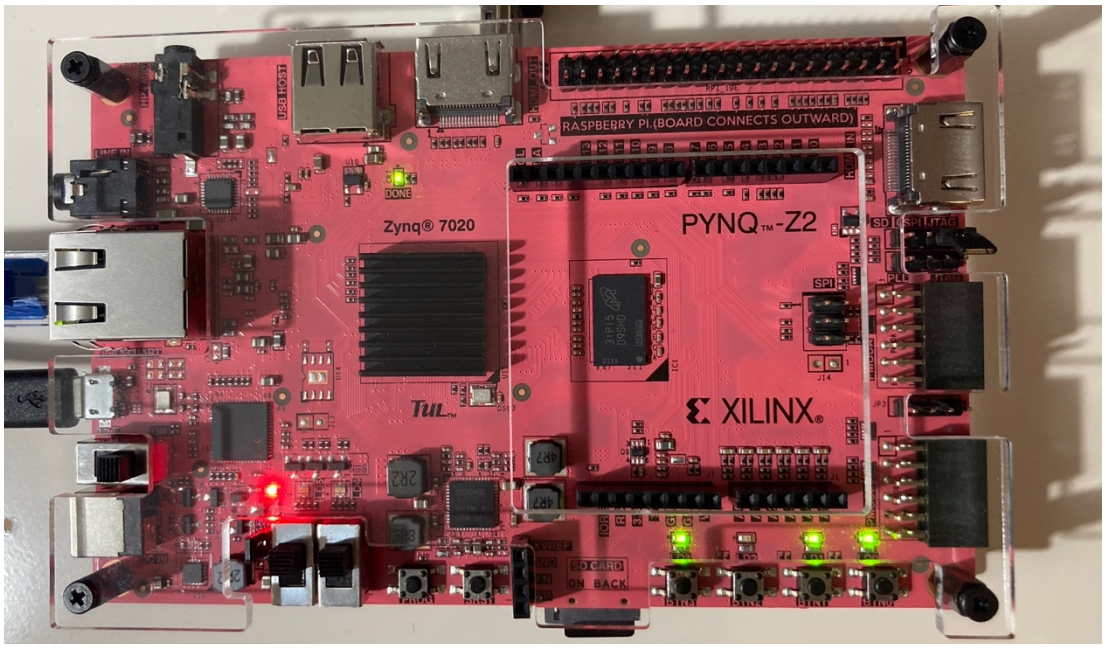

Figura 15 – Resultado final de la parte 3 del Tutorial

Los resultados de este proyecto demuestran cómo programar la FPGA de la placa PYNQ-Z2 utilizando Verilog y Vivado para interactuar con componentes físicos, en este caso, LEDs. Se muestra cómo a través de la creación de módulos en Verilog, se puede controlar el parpadeo de los LEDs basándose en la lógica de los botones y la utilización de divisores de reloj para modificar la frecuencia de parpadeo de los LEDs, evidenciando el poder y la flexibilidad del diseño de hardware con FPGA.

3.2.2 Utilización del puerto HDMI

La configuración de la interfaz HDMI en la plataforma PYNQ-Z2 para la transmisión de imágenes desde el SoC hacia un monitor exterior constituyó un paso crucial, implicando una interacción directa con los buffers de vídeo y la

administración efectiva de la señal HDMI. Este proceso no solo ofreció una inmersión inicial en el manejo práctico del procesamiento de vídeo a nivel de hardware, sino que también estableció una experiencia tangible en la manipulación de flujos de vídeo.

Por otro lado, el desarrollo de un script específico en Jupyter destinado a facilitar el envío de imágenes a través del puerto HDMI es clave para corroborar la capacidad del hardware para el procesamiento de vídeo. Dicha implementación sirvió para verificar que la plataforma Pynq-Z2 está equipada adecuadamente para gestionar operaciones intensivas de vídeo, proporcionando un sólido punto de partida para evaluaciones comparativas entre soluciones basadas en hardware y aquellas ejecutadas íntegramente en software. La exitosa transmisión de imágenes validó la operatividad del sistema bajo condiciones reales de uso y también resaltó su potencial para ser aplicado en proyectos avanzados de procesamiento de vídeo.

Explicación de la prueba:

El siguiente código carga un overlay en una placa PYNQ, lee una imagen llamada `sahara.jpg` usando `OpenCV`[12], configura la salida HDMI para mostrar la imagen con una resolución de 1920x1080 y 24 bits de color, redimensiona la imagen para que coincida con esta resolución, y finalmente muestra la imagen a través de HDMI. Además, establece el nivel de registro de `OpenCV` en silencioso para evitar mensajes de log innecesarios y libera los recursos de salida HDMI al final.

Código del archivo HDMI_connection.ipynb:

```
from pynq.overlays.base import BaseOverlay
from pynq.lib.video import *
from pynq import Overlay
import cv2
import os
import time

base = BaseOverlay("base.bit")   # Cargar el overlay

# Leer la imagen desde el archivo
image_path = 'sahara.jpg'
image = cv2.imread(image_path)
# Configurar la salida HDMI
Mode = VideoMode(1920, 1080, 24) # Modo del monitor según la imagen
hdmi_out = base.video.hdmi_out
hdmi_out.configure(Mode, PIXEL_BGR)
hdmi_out.start()
```

[12] OpenCV (Open Source Computer Vision Library) es una biblioteca de software de código abierto que incluye varios algoritmos de visión por computadora y procesamiento de imágenes. Se utiliza para aplicaciones como el reconocimiento facial, la detección de objetos, la clasificación de imágenes y la reconstrucción 3D. Está disponible para C++, Python, Java y otros lenguajes de programación, lo que facilita su integración en diferentes sistemas y aplicaciones.

```
# Redimensionar la imagen para que coincida con la resolución de salida
resized_image = cv2.resize(image, (1920, 1080))

# Establecer nivel de registro de OpenCV en silencioso
os.environ["OPENCV_LOG_LEVEL"] = "SILENT"

# Mostrar la imagen redimensionada a través de HDMI Out
outframe = hdmi_out.newframe()
outframe[0:1080, 0:1920, :] = resized_image
hdmi_out.writeframe(outframe)

# Liberar recursos
hdmi_out.stop()
hdmi_out.close()
```

Figura 16 – Resultado de la prueba de conexión HDMI

3.2.3 Primer diseño en Vitis HLS

La implementación del diseño "addmul" mediante Vitis HLS constituyó una experiencia introductoria esencial para entender la colaboración efectiva entre la herramienta de síntesis de alto nivel y la plataforma PYNQ-Z2. A pesar de su aparente simplicidad, este diseño desempeñó un papel clave en demostrar el flujo completo de desarrollo de una IP personalizada, abarcando desde la codificación inicial en C/C++ hasta su incorporación como un módulo operativo dentro de un entorno de sistema embebido.

La ejecución de "addmul" en Vitis HLS marcó un avance significativo en la comprensión de la síntesis de hardware de nivel superior. Esta experiencia inicial reveló cómo las operaciones básicas pueden ser eficientemente traducidas e implementadas en la lógica de hardware, ofreciendo una valiosa lección sobre la

optimización de algoritmos específicamente dirigida hacia el procesamiento en arquitecturas FPGA.

El siguiente diseño se ha basado en el proyecto realizado en [24].

Este apartado enfatiza el desarrollo y la integración de una IP personalizada en FPGA utilizando Vitis HLS y su posterior implementación en Vivado, culminando con la utilización de esta IP desde un entorno Python con Jupyter Notebook en la placa PYNQ-Z2. Este proceso es esencial para demostrar la transición desde la conceptualización de una IP, que realiza operaciones básicas de suma y multiplicación, hasta su implementación práctica en un dispositivo hardware.

Inicialmente, en Vitis HLS, creamos un nuevo proyecto y desarrollamos una IP denominada `addmul`, que realiza operaciones de suma y multiplicación. Este paso incluye la escritura del código en C/C++, que luego es sintetizado en RTL (Register Transfer Level), permitiendo su simulación y validación dentro de Vitis HLS. La creación de este proyecto y su código fuente son fundamentales, ya que establecen la base funcional de la IP que se integrará en el entorno de hardware.

Código de la IP desarrollada en Vitis HLS, archivo addmul.cpp:

```
void addmul(int a, int b, int& c, int& m) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE s_axilite port=a
#pragma HLS INTERFACE s_axilite port=b
#pragma HLS INTERFACE s_axilite port=c
#pragma HLS INTERFACE s_axilite port=m

	c = a + b;
	m = a * b;

}
```

Código de direcciones de memoria para la IP desarrollada en Vitis HLS, addmul.cpp. Ruta del archivo: addmul/solution1/impl/ip/drivers/addmul_v1_0/src/xaddmul_hw. Este código es útil si queremos llamar a las direcciones de memoria desde el programa que desarrollamos a un nivel superior, ya sea en Vivado, o en el caso de este ejemplo, en Jupyter, como se ve a continuación.

```
#define XADDMUL_CONTROL_ADDR_A_DATA 0x10
#define XADDMUL_CONTROL_BITS_A_DATA 32
#define XADDMUL_CONTROL_ADDR_B_DATA 0x18
#define XADDMUL_CONTROL_BITS_B_DATA 32
#define XADDMUL_CONTROL_ADDR_C_DATA 0x20
#define XADDMUL_CONTROL_BITS_C_DATA 32
#define XADDMUL_CONTROL_ADDR_C_CTRL 0x24
#define XADDMUL_CONTROL_ADDR_M_DATA 0x30
#define XADDMUL_CONTROL_BITS_M_DATA 32
#define XADDMUL_CONTROL_ADDR_M_CTRL 0x34
```

Una vez validada la IP en Vitis HLS mediante simulaciones, procedemos a exportarla para su uso en Vivado. Este proceso transforma la IP en un componente listo para integrarse en un diseño mayor. En Vivado, se crea un nuevo proyecto donde esta IP se importa y se añade a un diseño de bloque, conectándose con otros componentes, como el sistema procesador Zynq. Aquí, la asignación de pines y la configuración de interfaces son cruciales para asegurar la correcta comunicación entre la IP y el resto del sistema en la placa PYNQ-Z2.

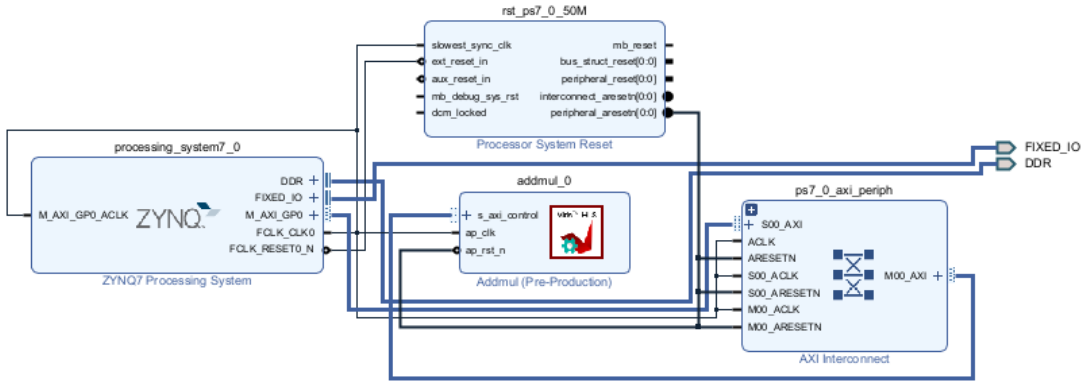

Figura 17 – Interconexión de las IPs en Vivado para el diseño hardware del addmul

Finalmente, con el diseño completado y el bitstream generado en Vivado, trasladamos la IP a la placa PYNQ-Z2 para su ejecución práctica. Utilizando Python en un entorno Jupyter Notebook, demostramos el funcionamiento de la IP personalizada, evidenciando la eficacia del proceso de diseño e implementación. Este paso es vital para verificar la funcionalidad de la IP en un entorno real, cerrando el ciclo de desarrollo desde la conceptualización hasta la aplicación práctica.

Código del archive addmul.ipynb:

```
from pynq import Overlay
overlay = Overlay('design_1_addmul.bit')

add_ip = overlay.addmul_0

# Damos un valor a la variable a = 4
add_ip.write(0x10, 4)
# Damos un valor a la variable b = 5
add_ip.write(0x18, 5)

# Leemos el valor de la suma
add_ip.read(0x20)
# Leemos el valor de la multiplicación
add_ip.read(0x30)
```

A lo largo de este tutorial, se destacan los pasos críticos en el proceso de desarrollo de IPs para FPGAs, desde la generación del código inicial en Vitis HLS hasta la implementación final en la placa PYNQ-Z2. Cada archivo generado en este proceso, ya sea código fuente, configuraciones de proyecto, o el bitstream final, cumple un rol específico en la creación de una solución de hardware efectiva, facilitando así la experimentación y el desarrollo de aplicaciones avanzadas.

3.2.4 Modificaciones en el overlay original

En el transcurso de nuestro viaje hacia el dominio del procesado de vídeo en hardware mediante el uso de sistemas SoC, nos enfocamos en la adaptación y personalización del diseño del hardware para satisfacer las demandas específicas de nuestras aplicaciones de procesado de vídeo. Esta sección destaca cómo, a través de un proceso meticuloso y detallado, transformamos los requerimientos funcionales en soluciones de hardware eficientes y optimizadas. Al abordar aspectos como la generación de módulos personalizados, la integración de IP en Vivado, y la utilización de herramientas como Vitis HLS y Jupyter, delineamos un marco general que guía desde la conceptualización inicial hasta la implementación final en la placa PYNQ-Z2. Se hace énfasis en situaciones en las que esta personalización es imprescindible, tales como el manejo de flujos de datos de vídeo en tiempo real, la optimización del rendimiento y la eficiencia energética, y la adaptación a requisitos de procesamiento específicos. A través de esta sección buscamos ilustrar los pasos generales seguidos en el desarrollo de hardware para aplicaciones de procesado de vídeo, y proporcionar una comprensión profunda de por qué y cómo se crean y utilizan los ficheros clave en este proceso.

En este ejemplo se modifica el bloque HDMI Out para obtener la aplicación deseada, en este caso cambiar el formato de la imagen de RGB a YUV. Como se ha mencionado en apartados anteriores, esta es solo una posible solución, hay infinitas formas de conseguir el resultado deseado modificando las partes que se desee del overlay original de la placa.

Modificar el bloque HDMI Out en Vivado para ajustar los píxeles con la IP "color_convert" demostró de manera avanzada la flexibilidad de aplicar funciones específicas sobre vídeos en tiempo real. Esta modificación muestra que al retocar el overlay original, es posible introducir toda una gama de filtros y funciones mediante la edición directa de los bloques IP. Esta habilidad para personalizar el tratamiento de vídeo directamente en el hardware abre muchas puertas para el desarrollo de aplicaciones avanzadas mediante SoCs y FPGAs.

La incorporación de esta IP en el overlay de la PYNQ implicó realizar cambios específicos en el diseño original, incluyendo la adición de la IP al esquema de bloques en Vivado, la configuración de interfaces de comunicación y la creación de un nuevo bitstream para su implementación en Jupyter. Estas modificaciones evidencian el amplio margen de personalización y expansión que ofrece el

sistema PYNQ-Z2, facilitando la adaptación del hardware a las exigencias concretas de cualquier proyecto.

Esta prueba aborda la modificación del overlay de la placa para incluir una IP personalizada de conversión de color en el flujo de HDMI Out. El objetivo es enriquecer el conocimiento práctico sobre diseño de sistemas embebidos en FPGAs, utilizando herramientas avanzadas como Vitis HLS y Vivado, y realizar pruebas reales en Jupyter Notebook. A continuación, se desarrollan más detalladamente cada uno de los puntos del proyecto.

Fase 1: Investigación y reconstrucción del overlay original

1. Investigación del Funcionamiento del Overlay Original: Para entender profundamente el overlay de la PYNQ Z2, es fundamental reconstruir el proyecto a partir de los documentos originales disponibles en la memoria SD de la placa o mediante la descarga de los archivos del proyecto Open Source desde GitHub [25].

2. Recreación del Overlay mediante la Consola TCL de Vivado: Utilizando Vivado, el overlay se puede recrear ejecutando los scripts en la consola del programa. Esto evitará tener que añadir todas las IPs de una en una, simplificando enormemente el proceso. Este método restablece el diseño original en el entorno de Vivado, permitiendo una inspección detallada y facilitando la comprensión de su arquitectura y funcionamiento interno. Se siguieron los siguientes pasos:

    a. Una vez creado el proyecto, se escribió en la línea de comandos la instrucción para acceder a la carpeta con los archivos .tcl necesarios para reconstruir el overlay original: `cd {PYNQ/boards/Pynq-Z2/base}`
    b. Después, cuando se accedió a la carpeta, se ejecutó la instrucción para importar todas las IPs que utiliza este overlay, esta información está en el archivo build_ip.tcl de la carpeta base. Por lo tanto, se escribió en la línea de comandos: `source build_ip.tcl`

    c. Finalmente, con las IPs importadas y en la carpeta base, se recrea toda el overlay mediante el comando: `source base.tcl`. Este proceso de interconexión suele tardar unos 10 minutos e incluye la validación del diseño una vez interconectadas las IPs.

Fase 2: Estudio de la arquitectura y modificación

1. Estudio Detallado de la Arquitectura y el Flujo HDMI Out: Con el overlay original recreado, se realiza un análisis exhaustivo para comprender la arquitectura y el flujo de datos, especialmente en lo que respecta al procesamiento y salida de vídeo a través del bloque HDMI Out. Este análisis revela que después del bloque de conversión de color (`color_convert`), hay una oportunidad para insertar una IP de conversión de color adicional

que procese los píxeles individualmente antes de su transmisión al puerto HDMI Out.

2. Desarrollo de la IP de Conversión de Color en Vitis HLS: En Vitis HLS, se desarrolla la IP `color_conversion` con la capacidad de ser compatible y funcionar eficientemente dentro del flujo de datos del bloque HDMI Out. Este desarrollo implica la creación de un código que pueda procesar cada píxel individualmente, aplicando la conversión de color deseada antes de su salida por el HDMI, concretamente se ha pasado de una gama de colores RGB a YUV mediante aritmética de punto fijo aplicada pixel a pixel, cada uno de ellos de 24 bits.

Código de cabecera de la IP, archivo color_conversion.hpp:

```
#include <ap_int.h>
#include "hls_stream.h"
#include <ap_axi_sdata.h>

typedef ap_uint<8> pixel_type;
typedef ap_int<8> pixel_type_s;
typedef ap_axiu<24,1,1,1> pixel_data;
typedef hls::stream<pixel_data> video_stream;

void rgb2yuv(video_stream& stream_in, video_stream& stream_out);
```

Código del programa principal de la IP, archivo color_conversion.cpp:

```
#include "color_conversion.hpp"

// Definiendo tipos de datos de punto fijo
typedef ap_fixed<16, 8> fixed_type;     // 16 bits total, 8 bits parte entera

void rgb2yuv(video_stream& stream_in, video_stream& stream_out) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=stream_in
#pragma HLS INTERFACE axis port=stream_out

#pragma HLS pipeline II=1

    pixel_data curr_pixel;
    pixel_type r, g, b;
    fixed_type y, u, v;

    // Constantes precalculadas para la conversión
    const fixed_type Kr = 0.299;
    const fixed_type Kg = 0.587;
    const fixed_type Kb = 0.114;
    const fixed_type Ku = -0.147;
    const fixed_type Kv = 0.615;
    const fixed_type offset = 128;

    while (!stream_in.empty()) {
        #pragma HLS loop_tripcount min=1 max=1920    // Ajustar tamaño imagen
        stream_in.read(curr_pixel);
```

```
        // Extraer componentes RGB
        r = curr_pixel.data(23, 16);
        g = curr_pixel.data(15, 8);
        b = curr_pixel.data(7, 0);

        // Conversión de RGB a YUV con aritmétiica de punto fijo
        y = Kr * r + Kg * g + Kb * b;
        u = Ku * r - Kg * g + Kb * b + offset;
        v = Kv * r - Kg * g - Kb * b + offset;

        // Combinar los componentes YUV en un solo píxel
        curr_pixel.data = (v, u, y);

        stream_out.write(curr_pixel);
    }

}
```

Fase 3: Integración y validación en Vivado

1. Exportación e Importación de la IP desde Vitis HLS a Vivado: Una vez que la IP cumple con las restricciones de tiempo y funciona según lo previsto en las simulaciones de Vitis HLS, se exporta a Vivado. Este proceso incluye la generación de un paquete IP que se puede integrar fácilmente en el diseño existente en Vivado.

2. Interconexión con los Bloques IP del HDMI Out: En Vivado, se procede a interconectar la nueva IP `color_conversion` con los bloques existentes del HDMI Out, prestando especial atención a la compatibilidad de los interfaces y a la correcta gestión del flujo de datos.

3. Validación del Diseño y Creación del Archivo de Restricciones: Se valida el diseño modificado para asegurar que no existan conflictos o problemas de funcionamiento con los demás bloques IP. Además, se crea y configura el archivo de restricciones necesario para el correcto mapeo físico de las señales y se genera del HDL Wrapper.

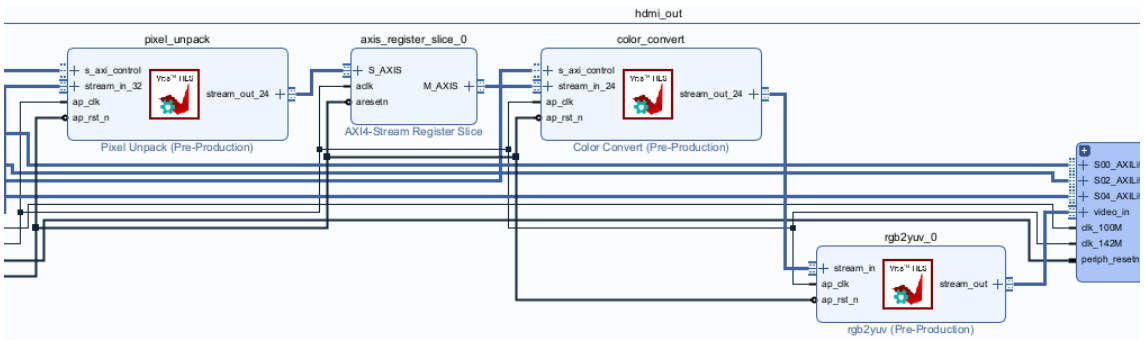

Figura 18 – Interconexión del nuevo bloque IP (`rgb2yuv`) con el bloque HDMI Out

Fase 4: Simulación, implementación y pruebas

1. Simulación, Implementación y Generación del Archivo `.bitstream`: Antes de la implementación física, se realiza una simulación completa del diseño modificado para verificar su funcionamiento. Tras confirmar que el diseño es sólido, se procede a la implementación y a la generación del archivo `.bitstream`, que encapsula el diseño para su carga en la FPGA.

2. Carga en Jupyter y Pruebas Funcionales: Con el archivo `.bitstream` generado, este se carga en la placa PYNQ Z2 a través de un entorno Jupyter Notebook. Se desarrolla y ejecuta un script en Python que permite probar la nueva funcionalidad de conversión de color, evaluando el impacto de la IP personalizada en el flujo de vídeo HDMI Out.

Código del archivo Test_color_inverter.ipynb:

```
from pynq.overlays.base import BaseOverlay
from pynq.lib.video import *
from pynq import Overlay
import cv2
import os
import time

# Cargar el overlay
base = BaseOverlay("base_wrapper.bit")

# Configurar la salida HDMI
Mode = VideoMode(640, 480, 24)   # Modo del monitor: 640x480 @ 60Hz
hdmi_out = base.video.hdmi_out
hdmi_out.configure(Mode, PIXEL_BGR)
hdmi_out.start()


# Configuración de la cámara (entrada)
frame_in_w, frame_in_h = 640, 480

# Establecer nivel de registro de OpenCV en silencioso
os.environ["OPENCV_LOG_LEVEL"] = "SILENT"

# Liberar cualquier recurso de cámara existente
cv2.VideoCapture(0).release()

start_time = time.time()
num_frames = 0
read_errors = 0

try:
    # Inicializar la cámara
    videoIn = cv2.VideoCapture(0)
    videoIn.set(cv2.CAP_PROP_FRAME_WIDTH, frame_in_w)
    videoIn.set(cv2.CAP_PROP_FRAME_HEIGHT, frame_in_h)

    while (base.buttons[3].read()==0):
        # Capturar video frame por frame
        ret, frame_vga = videoIn.read()

        if ret:
            # Incrementar el contador de frames
            num_frames += 1
```

```
            # Aquí se aplica el procesamiento de imagen si es necesario
            processed_frame = frame_vga

            # Mostrar el frame procesado a través de HDMI Out
            outframe = hdmi_out.newframe()
            outframe[0:frame_in_h, 0:frame_in_w, :] = processed_frame
            hdmi_out.writeframe(outframe)

        else:
            # Incrementar el contador de errores de lectura
            read_errors += 1
            print("Error al leer de la cámara.")
            # break

except Exception as e:
    print(f"Ha ocurrido un error: {e}")

finally:
    # Calcular el tiempo transcurrido
    end_time = time.time()
    elapsed_time = end_time - start_time

    # Calcular los frames por segundo
    fps = (num_frames - read_errors) / elapsed_time

    # Mostrar resultados
    print("Fin de la prueba")
    print("Frames por segundo: " + str(fps))
    print("Número de errores: " + str(read_errors))
    # Liberar recursos
    videoIn.release()
    hdmi_out.stop()
    hdmi_out.close()
```

Resultado del código Jupyter:

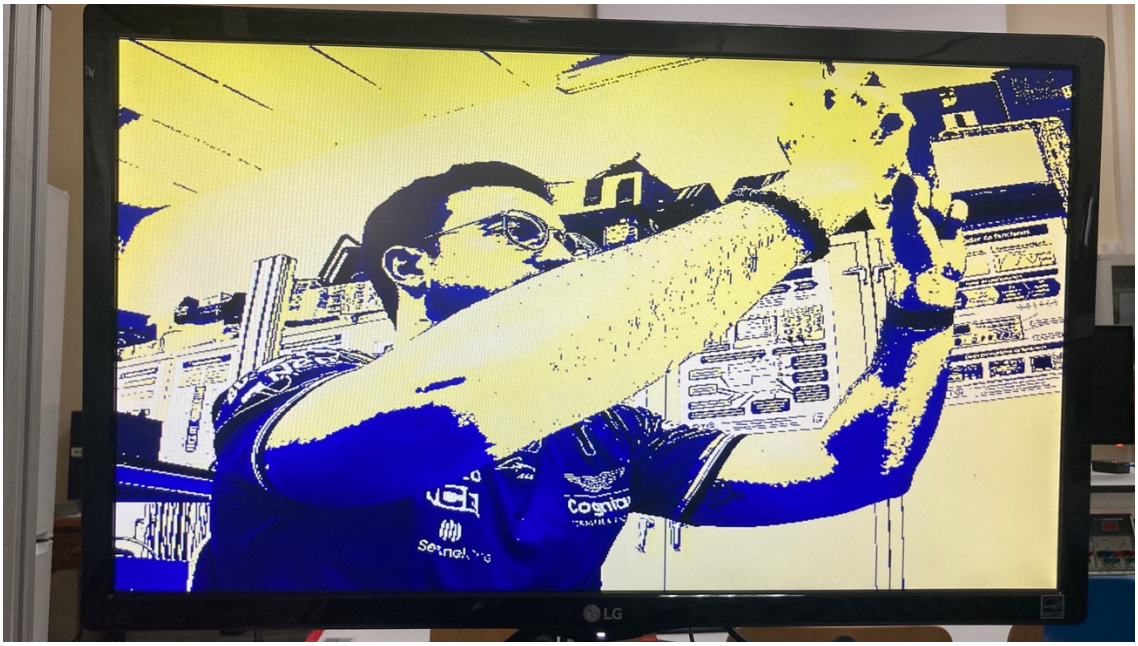

Figura 19 – Imagen generada con la escala de colores YUV en lugar de RGB

```
Fin de la prueba
Frames por segundo: 11.499096544449985
Número de errores: 0
```

Figura 20 – Resultado de la prueba Test_color_inverter.ipynb

3.3 Conclusiones

Cada una de estas pruebas fue cuidadosamente seleccionada para construir progresivamente una comprensión detallada y práctica del procesamiento de vídeo en hardware, utilizando el ecosistema PYNQ-Z2. La secuencia de pruebas no solo justifica la metodología de aprendizaje adoptada, sino que también establece una base sólida para el desarrollo de aplicaciones de procesamiento de vídeo en hardware más complejas y optimizadas.

Estas pruebas iniciales, además de validar la capacidad y versatilidad del hardware y software, también sirve para inspirar la exploración de técnicas avanzadas de procesamiento de aplicaciones sobre FPGAs y SoCs. La progresión desde la familiarización básica hasta la implementación de procesos complejos de procesamiento de vídeo demuestra la escalabilidad y versatilidad del ecosistema PYNQ-Z2 para aplicaciones de ingeniería.

4 PRUEBAS SOFTWARE

Una vez realizadas las primeras pruebas que han permitido poner en pie el flujo de diseño y comprobar que los resultados son los correctos, se plantea una segunda fase dedicada a la evaluación comparativa entre los enfoques de procesado de vídeo en software y en hardware. El propósito de esta fase es doble: por un lado, busca establecer las bases para una comprensión profunda de las ventajas y limitaciones inherentes al procesado de vídeo mediante software; por otro lado, aspira a generar un conjunto de datos empíricos que permitan una valoración objetiva de las diferencias de rendimiento, eficiencia en el uso de recursos y calidad del resultado final entre ambos enfoques.

La relevancia de esta comparativa radica en su capacidad para iluminar decisiones de diseño y optimización en el ámbito del procesado de vídeo, un campo de estudio y aplicación que ha cobrado especial importancia en la era digital actual. Con una creciente demanda de aplicaciones capaces de procesar vídeo en tiempo real para una variedad de propósitos, la elección entre implementaciones de software y de hardware no es meramente técnica, sino que conlleva implicaciones significativas en términos de costos, escalabilidad y accesibilidad.

Este capítulo del TFG se centra en el desarrollo y la ejecución de un conjunto de pruebas específicas sobre implementaciones de software de diversos filtros de procesado de vídeo. Dichas pruebas, basadas en los notebooks de Jupyter "**Test-Negative.ipynb**", "**Test-Sepia.ipynb**", y "**Test-USB2HDMI_Sobel.ipynb**", se diseñan para emular las funcionalidades implementadas previamente en hardware, con el fin de obtener una comparativa directa que abarque el rendimiento, el consumo de recursos, y también la fidelidad y la calidad de los vídeos procesados.

La importancia de este análisis comparativo se extiende más allá de los límites académicos de este TFG. Se busca contribuir a la base de conocimiento existente con aportaciones prácticas y recomendaciones específicas que faciliten la toma de decisiones en proyectos relacionados con el procesado de vídeo, tanto en plataformas de hardware especializado como en soluciones basadas puramente en software.

4.1 Metodología de pruebas

La metodología de pruebas adoptada para la evaluación comparativa entre las implementaciones de procesado de vídeo en software y en hardware se estructura en torno a varios ejes fundamentales. Estos ejes abarcan la selección de herramientas y librerías, la definición de criterios de comparación, y el diseño de un protocolo de pruebas coherente y replicable. A continuación, se detalla cada uno de estos aspectos.

4.1.1 Selección de herramientas y librerías

Para las pruebas de software, se optó por seguir utilizando un entorno de desarrollo basado en Python, dada su amplia adopción en la comunidad científica y técnica para tareas de procesamiento de imágenes y vídeo. Las librerías específicas incluyen:

- OpenCV: Utilizada para el procesamiento de imágenes y vídeos, permitiendo la implementación de filtros y técnicas de visión por ordenador.

- NumPy: Empleada para la manipulación eficiente de matrices, que constituyen la base de las operaciones de procesado de imágenes y vídeo.

- Jupyter Notebook: Seleccionado como plataforma de desarrollo y documentación, facilitando la integración de código, resultados y análisis en un único documento interactivo y que como ya se ha visto se encuentra disponible en la plataforma de desarrollo Pynq-Z2 que es la que estamos utilizando.

4.1.2 Criterios de comparación

Los criterios seleccionados para la comparación entre las implementaciones software y hardware son:

1. Tiempo de Ejecución: Medición del tiempo que tarda en procesarse un vídeo, desde la carga hasta la generación del resultado final.

2. Uso de Recursos: Evaluación del consumo de recursos del sistema, como el uso de CPU y memoria durante el procesamiento.

3. Calidad del Procesado: Análisis cualitativo y cuantitativo de la calidad del vídeo resultante, considerando factores como la fidelidad del filtro aplicado y la preservación de detalles.

4.1.3 Protocolo de pruebas

El protocolo de pruebas se diseñó con el objetivo de asegurar la replicabilidad y la objetividad de los resultados. Los pasos incluyen:

1. Preparación del Entorno de Pruebas:

    - Asegurar condiciones consistentes de hardware y software para todas las pruebas.

- El diseño del código Jupyter debe ser lo más similar posible para las pruebas software y hardware.
- Seleccionar un conjunto representativo de vídeos de prueba, variando en resolución, duración y características visuales.

2. Ejecución de Pruebas en Software:
   - Implementar cada filtro (Negativo, Sepia, Sobel) en el entorno de software, utilizando las librerías seleccionadas.
   - Medir el tiempo de ejecución y el uso de recursos para cada vídeo de prueba.
   - Evaluar la calidad del vídeo procesado mediante criterios predefinidos.
3. Recopilación de Datos y Análisis:
   - Registrar los resultados de tiempo de ejecución, uso de recursos y evaluaciones de calidad.
   - Preparar análisis comparativos con los resultados obtenidos de las pruebas en hardware.

Este enfoque metodológico busca proporcionar una base sólida para la evaluación comparativa, permitiendo extraer conclusiones relevantes sobre la eficacia y eficiencia del procesado de vídeo tanto en software como en hardware.

4.2 Desarrollo de pruebas software

El desarrollo de las pruebas software se centra en la evaluación de tres tipos de filtros de procesado de vídeo: Negativo, Sepia y Sobel. Utilizando Python y librerías especializadas como OpenCV y NumPy, se implementaron estos filtros en el entorno de Jupyter Notebook. A continuación, se detalla el proceso y los resultados obtenidos para cada uno.

4.2.1 Filtro Negativo

El filtro Negativo invierte los colores del vídeo, transformando cada pixel según la fórmula: `NuevoValor = 255 - ValorActual`. Este filtro es útil para resaltar detalles ocultos en regiones oscuras o sobreexpuestas.

Implementación

Utilizando OpenCV, se leyó cada frame del vídeo de prueba y se aplicó la transformación de inversión de color.

Código de la parte software del archivo Test-Negativo.ipyb:

Se puede encontrar la función que aplica el filtro negativo a cada frame definido en la función `apply_negative`.

```
'''
Software
'''
import cv2
import numpy as np
from pynq.overlays.base import BaseOverlay
from pynq.lib.video import *
import time
import psutil

# Función para medir la CPU y la memoria utilizada
def measure_resources():
    cpu_percent = psutil.cpu_percent()
    memory_percent = psutil.virtual_memory().percent
    return cpu_percent, memory_percent
# Cargar el overlay base
base = BaseOverlay("base.bit")

# Configuración de la salida HDMI
hdmi_out = base.video.hdmi_out
hdmi_out.configure(VideoMode(640, 480, 24), PIXEL_BGR)
hdmi_out.start()

# Liberar cualquier recurso de cámara existente
cv2.VideoCapture(0).release()

# Inicializar la cámara
cap = cv2.VideoCapture(0)
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)

# Función para aplicar el filtro negativo a un fotograma
def apply_negative(frame):
    # Convertir la imagen a negativo
    negative_frame = 255 - frame
    return negative_frame

# Inicialización de variables para el cálculo de frames por segundo
start_time = time.time()
num_frames = 0
read_errors = 0
total_cpu_usage = 0
total_memory_usage = 0

while (base.buttons[3].read()==0):
    fstime=time.time()
    ret, frame = cap.read()
    if ret:
        # Incrementar el contador de frames
        num_frames += 1
        # Aplicar el filtro negativo
        negative_frame = apply_negative(frame)
```

```
        # Crear un nuevo objeto Frame
        frame_pynq = hdmi_out.newframe()
        # Copiar los datos del marco OpenCV al objeto Frame
        np.copyto(frame_pynq, negative_frame)
        # Mostrar la imagen en el monitor HDMI Out
        cv2.putText(frame_pynq,"FPS:"+str(round(1/(time.time()-
fstime),4)),(50,25),0,0.8,(255,255,255),1)
        hdmi_out.writeframe(frame_pynq)

        # Medir recursos después de aplicar el filtro
        cpu_usage, memory_usage = measure_resources()
        total_cpu_usage += cpu_usage
        total_memory_usage += memory_usage
    else:
        # Incrementar el contador de errores de lectura
        read_errors += 1
        print("Error al leer de la cámara.")

# Calcular el tiempo transcurrido
end_time = time.time()
elapsed_time = end_time - start_time

# Calcular los frames por segundo
fps = (num_frames - read_errors) / elapsed_time

# Calcular el promedio de uso de CPU y memoria
avg_cpu_usage = total_cpu_usage / num_frames
avg_memory_usage = total_memory_usage / num_frames

# Mostrar resultados
print("Programa finalizado")
print("Frames por segundo: " + str(fps))
print("Número de errores leídos: " + str(read_errors))
print("Uso promedio de CPU: " + str(avg_cpu_usage) + "%")
print("Uso promedio de memoria: " + str(avg_memory_usage) + "%")

# Liberar recursos
cap.release()
hdmi_out.stop()
hdmi_out.close()
```

Resultados

A raíz del código anterior, específicamente de las funciones de la librería `psutil`, se ha creado una función llamada `measure_resources`, la cual engloba las funciones `psutil.cpu_percent()` y `psutil.virtual_memory().percent` que proporcionan los resultados del uso promedio de CPU y de memoria respectivamente. Se han realizado 5 intentos por cada tipo de filtro, obteniendo los siguientes resultados:

N.º de Intento	1	2	3	4	5
Frames por segundo	11.8927	12.2085	12.2170	11.8929	11.9718
Número de errores leídos	0	0	0	0	0
Uso promedio de CPU	26.6382%	33.6833%	29.3671%	28.9915%	32.2249%
Uso promedio de memoria	59.3131%	58.1083%	60.7842%	54.0340%	60.8463%

Tabla 1 – Resultados de aplicar un filtro negativo en software

4.2.2 Filtro Sepia

El efecto Sepia tiñe el vídeo con un color marrón-rojizo, imitando la apariencia de fotografías antiguas. Este filtro agrega un toque nostálgico a las imágenes.

Implementación

La aplicación del filtro Sepia requirió una conversión de color a escala de grises seguida de la aplicación de una matriz específica de tono Sepia sobre los valores de los píxeles.

Código de la parte software del archivo Test-Sepia.ipyb:

Se puede encontrar la función que aplica el filtro sepia a cada frame definido en la función `apply_sepia`.

```
'''
Software
'''
import cv2
import numpy as np
from pynq.overlays.base import BaseOverlay
from pynq.lib.video import *
```

```
import time
import psutil

# Función para aplicar el filtro sepia a cada frame
def apply_sepia(frame):
    # Aplicar el efecto sepia manualmente
    sepia_filter = np.array([[0.393, 0.769, 0.189],
                             [0.349, 0.686, 0.168],
                             [0.272, 0.534, 0.131]])
    sepia_frame = cv2.transform(frame, sepia_filter)
    sepia_frame = np.clip(sepia_frame, 0, 255).astype(np.uint8)
    return sepia_frame

# Función para medir la CPU y la memoria utilizada
def measure_resources():
    cpu_percent = psutil.cpu_percent()
    memory_percent = psutil.virtual_memory().percent
    return cpu_percent, memory_percent

# Cargar el overlay base
base = BaseOverlay("base.bit")

# Configuración de la salida HDMI
hdmi_out = base.video.hdmi_out
hdmi_out.configure(VideoMode(640, 480, 24), PIXEL_BGR)
hdmi_out.start()

# Liberar cualquier recurso de cámara existente
cv2.VideoCapture(0).release()

# Inicializar la cámara
cap = cv2.VideoCapture(0)
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)

# Inicialización de variables para el cálculo de frames por segundo
start_time = time.time()
num_frames = 0
read_errors = 0
total_cpu_usage = 0
total_memory_usage = 0

while (base.buttons[3].read()==0):
    fstime=time.time()
    ret, frame = cap.read()
    if ret:
        # Incrementar el contador de frames
        num_frames += 1
        # Aplicar el filtro sepia
        sepia_frame = apply_sepia(frame)
        # Crear un nuevo objeto Frame
        frame_pynq = hdmi_out.newframe()
        # Copiar los datos del marco OpenCV al objeto Frame
        np.copyto(frame_pynq, sepia_frame)
        # Mostrar la imagen en el monitor HDMI Out
        cv2.putText(frame_pynq,"FPS:"+str(round(1/(time.time()-
fstime),4)),(50,25),0,0.8,(255,255,255),1)
        hdmi_out.writeframe(frame_pynq)

        # Medir recursos después de aplicar el filtro
        cpu_usage, memory_usage = measure_resources()
        total_cpu_usage += cpu_usage
        total_memory_usage += memory_usage
    else:
        # Incrementar el contador de errores de lectura
        read_errors += 1
        print("Error al leer de la cámara.")
```

```
# Calcular el tiempo transcurrido
end_time = time.time()
elapsed_time = end_time - start_time

# Calcular los frames por segundo
fps = (num_frames - read_errors) / elapsed_time

# Calcular el promedio de uso de CPU y memoria
avg_cpu_usage = total_cpu_usage / num_frames
avg_memory_usage = total_memory_usage / num_frames

# Mostrar resultados
print("Prueba finalizada")
print("Frames por segundo: " + str(fps))
print("Número de errores leídos: " + str(read_errors))
print("Uso promedio de CPU: " + str(avg_cpu_usage) + "%")
print("Uso promedio de memoria: " + str(avg_memory_usage) + "%")

# Liberar recursos
cap.release()
hdmi_out.stop()
hdmi_out.close()
```

Resultados

En la tabla 2 se muestran los resultados obtenidos para este filtro, utilizando para ello las mismas funciones especificadas anteriormente de la librería `psutil`.

N.º de Intento	1	2	3	4	5
Frames por segundo	8.9993	9.1438	9.2927	9.2873	9.3750
Número de errores leídos	0	0	0	0	0
Uso promedio de CPU	61.3549%	56.5452%	56.3512%	56.3890%	56.8657%
Uso promedio de memoria	62.6431%	62.6426%	62.6856%	63.6625%	63.5090%

Tabla 2 – Resultados de aplicar un filtro sepia en software

4.2.3 Filtro Sobel para detección de bordes

El filtro Sobel es utilizado para la detección de bordes en imágenes, resaltando las transiciones de intensidad. Es fundamental en aplicaciones de visión por computador para el análisis de estructuras y formas.

Este filtro Sobel se ha basado en el proyecto Open Source [26]. El estudio de este proyecto, junto con las aportaciones propias que pueden verse en la parte del diseño hardware, ofrecen otra visión de cómo funciona la sinergia entre Hardware y Software para obtener mejores resultados.

Implementación

Se aplicó el operador Sobel sobre los frames del vídeo para calcular el gradiente de intensidad en las direcciones X e Y, combinando luego estos gradientes para obtener la magnitud de los bordes.

Código de la parte software del archivo Test-USB2HDMI_Sobel.ipyb:

En este código se llama a la función de la librería OpenCV `cv2.Laplacian` para aplicar el filtro sobel.

```
import cv2, time, warnings
import numpy as np
from pynq.lib.video import *
warnings.filterwarnings('ignore')
import pynq
from pynq.overlays.base import BaseOverlay
from pynq import Overlay
from pynq import allocate
import psutil

# Función para medir la CPU y la memoria utilizada
def measure_resources():
    cpu_percent = psutil.cpu_percent()
    memory_percent = psutil.virtual_memory().percent
    return cpu_percent, memory_percent

overlay = BaseOverlay('base_w_sobel.bit')

sobel = overlay.sobel_0
dma = overlay.axi_dma_0

def reset_all_dma():
    def dma_reset(dma):
        dma.sendchannel.stop()
        dma.recvchannel.stop()
        dma.sendchannel.start()
        dma.recvchannel.start()
    dma_reset(dma)
reset_all_dma()

resolution=(640,480)
padd=int((max(resolution)-min(resolution))/2)
recive_head=0+padd
recive_tail=recive_head+min(resolution)
buffer_shape=max(resolution)

input_buffer = allocate(shape=(buffer_shape,buffer_shape), dtype=np.uint8)
output_buffer = allocate(shape=(buffer_shape,buffer_shape), dtype=np.uint8)

def run_sobel():
    reset_all_dma()
    dma.sendchannel.transfer(input_buffer)
    dma.recvchannel.transfer(output_buffer)
    sobel.write(0x10,buffer_shape)
    sobel.write(0x18,buffer_shape)
    sobel.write(0x00,0x81) #start
    dma.sendchannel.wait()
    dma.recvchannel.wait()

frame_in_w,frame_in_h = resolution
cap = cv2.VideoCapture(0)
cap.set(cv2.CAP_PROP_FRAME_WIDTH, frame_in_w);
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, frame_in_h);
cap.isOpened()
```

```
'''
Usando OpenCV (sin acelerador PL)
'''
frame_out_w,frame_out_h = resolution
Mode = VideoMode(frame_out_w,frame_out_h,24)
hdmi_out = overlay.video.hdmi_out
hdmi_out.configure(Mode,PIXEL_BGR)
hdmi_out.start()

start=time.time()
num_frames=0
readError=0
total_cpu_usage = 0
total_memory_usage = 0
while (overlay.buttons[3].read()==0):
    fstime=time.time()
    ret, frame_vga = cap.read()
    if (ret):
        outframe = hdmi_out.newframe()
        laplacian_frame = cv2.Laplacian(frame_vga, cv2.CV_8U,ksize=3,
dst=outframe)
        cv2.putText(outframe,"FPS:"+str(round(1/(time.time()-
fstime),4)),(10,20),0,0.8,(255,255,255),1)
        hdmi_out.writeframe(outframe)
        num_frames+=1
        # Medir recursos después de procesar el frame
        cpu_usage, memory_usage = measure_resources()
        total_cpu_usage += cpu_usage
        total_memory_usage += memory_usage
    else:
        readError += 1
        print("Error al leer de la cámara.")

end = time.time()
print("Prueba finalizada")
print("Frames por segundo: " + str((num_frames-readError) / (end - start)))
print("Uso promedio de CPU: " + str(total_cpu_usage / num_frames) + "%")
print("Uso promedio de Memoria: " + str(total_memory_usage / num_frames) +
"%")
print("Número de errores leídos: " + str(readError))
```

Resultados

En la tabla 3 se muestran los resultados obtenidos para este filtro, utilizando para ello las mismas funciones especificadas anteriormente de la librería `psutil`.

N.º de Intento	1	2	3	4	5
Frames por segundo	9.6581	9.7458	9.4564	9.8303	9.4513
Número de errores leídos	0	0	0	0	0
Uso promedio de CPU	56.0400%	57.0178%	55.1263%	61.1178%	56.1521%
Uso promedio de memoria	49.6552%	49.6800%	57.8158%	47.6122%	47.4275%

Tabla 3 – Resultados de aplicar un filtro Sobel en software

4.3 Análisis de resultados

Los resultados de las pruebas software ofrecen una visión integral del rendimiento y la calidad del procesado de vídeo mediante filtros aplicados en software. Se observaron diferencias significativas en el tiempo de ejecución y el uso de recursos entre los filtros, lo cual es indicativo de la diferencia en la complejidad computacional de cada operación de procesado. La calidad del procesado, por su parte, fue consistentemente alta, demostrando la capacidad de las implementaciones software para lograr resultados visuales satisfactorios.

Como puede comprobarse, la complejidad de los tres filtros se refleja en el uso promedio de recursos de cada uno, por ejemplo, en cuanto Frames por segundo, la simplicidad del filtro negativo permite alcanzar alrededor de los 12 FPS, a diferencia de los otros dos que oscilan entre los 9.5 FPS. Por otro lado, con el uso de CPU sucede algo parecido, el negativo usa en torno el 30% y los otros un 57%. Esta diferencia no se refleja en lo que a uso de memoria se refiere, ya

que el del filtro negativo está en torno al 58.6%, el sepia utiliza un 63% y el sobel es el más eficiente en este aspecto con un uso del 50.4% aproximadamente.

Estos resultados son comparados con las implementaciones en hardware en el capítulo 6, para obtener una comprensión completa de las ventajas y limitaciones, permitiendo así recomendaciones justificadas para la selección de la plataforma de procesado más adecuada según los requisitos específicos de la aplicación.

5 PRUEBAS HARDWARE

5.1 Desarrollo de filtros de vídeo en Vitis HLS

5.1.1 Diseño y optimización de filtros

El desarrollo y optimización de filtros de vídeo para procesamiento en hardware se llevó a cabo en Vitis HLS, un entorno que permite la implementación de algoritmos de alto nivel en lenguaje C++ para hardware. Esto facilita la creación de IP (Intellectual Property) personalizadas para FPGAs. A continuación, se describen los pasos específicos tomados para cada filtro diseñado.

5.1.1.1 Negativo

El filtro Negativo invierte los colores de la imagen, generando un efecto visual donde las áreas claras se vuelven oscuras y viceversa. La implementación en Vitis HLS consistió en leer cada píxel del flujo de entrada, invertir los valores de sus componentes RGB y escribir el resultado en el flujo de salida. La optimización se centró en minimizar la latencia mediante la directiva `#pragma HLS pipeline`, permitiendo el procesamiento de un píxel por ciclo de reloj.

Código de cabecera HLS:

```
#include <ap_int.h>
#include "hls_stream.h"
#include <ap_axi_sdata.h>

typedef ap_uint<8> pixel_type;
typedef ap_axiu<24,1,1,1> pixel_data;
typedef hls::stream<pixel_data> video_stream;

void rgb2negative(video_stream& stream_in, video_stream& stream_out);
```

Código del programa principal HLS:

```
#include "negative_filter.hpp"

void rgb2negative(video_stream& stream_in, video_stream& stream_out) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=stream_in
#pragma HLS INTERFACE axis port=stream_out

#pragma HLS pipeline II=1

    pixel_data curr_pixel;
    pixel_type r, g, b;
    pixel_type new_r, new_g, new_b;

    while (!stream_in.empty()) {
        stream_in.read(curr_pixel);
```

```
        // Extraer componentes RGB
        r = curr_pixel.data(23, 16);
        g = curr_pixel.data(15, 8);
        b = curr_pixel.data(7, 0);

        // Aplicar filtro de negativo a los componentes RGB
        new_r = 255 - r;
        new_g = 255 - g;
        new_b = 255 - b;

        // Combinar los componentes en un solo pixel
        curr_pixel.data = (new_b, new_g, new_r);

        stream_out.write(curr_pixel);
    }
}
```

5.1.1.2 Sepia

El filtro Sepia aplica una transformación que da a la imagen un tono similar al de las fotografías antiguas. La implementación realizó una ponderación específica de los componentes RGB de cada píxel basándose en coeficientes que imitan el efecto sepia.

Se aplicaron optimizaciones para mejorar el rendimiento, como la utilización de aproximaciones enteras para los coeficientes y el desplazamiento de bits para realizar la multiplicación, mejorando la eficiencia al evitar el uso de operaciones en punto flotante. Este cambio principal con respecto al filtro original fue producto de que, debido a los cálculos y transformaciones necesarias para obtener el resultado deseado en cada píxel, no se cumplían las restricciones de tiempo impuestas para el componente en el bloque HDMI Out, dando fallos a la hora de la visualización en el monitor.

Código de cabecera HLS:

```
#include <ap_int.h>
#include "hls_stream.h"
#include <ap_axi_sdata.h>

typedef ap_uint<8> pixel_type;
typedef ap_axiu<24,1,1,1> pixel_data;
typedef hls::stream<pixel_data> video_stream;

void rgb2sepia(video_stream& stream_in, video_stream& stream_out);
```

Código del programa principal HLS:

```
#include "sepia_filter.hpp"

void rgb2sepia(video_stream& stream_in, video_stream& stream_out) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=stream_in
#pragma HLS INTERFACE axis port=stream_out
#pragma HLS pipeline II=1

    pixel_data curr_pixel;
    pixel_type r, g, b;
    pixel_type new_r, new_g, new_b;

    while (!stream_in.empty()) {
        stream_in.read(curr_pixel);

        // Extraer componentes RGB
        r = curr_pixel.data(23, 16);
        g = curr_pixel.data(15, 8);
        b = curr_pixel.data(7, 0);

        // Optimización: Uso de aproximaciones enteras y desplazamientos
        // Coeficientes aproximados y convertidos a enteros
        int coef_r_r = 40; // Aproximación de 0.393 * 102
        int coef_r_g = 78; // Aproximación de 0.769 * 102
        int coef_r_b = 19; // Aproximación de 0.189 * 102

        int coef_g_r = 36; // Aproximación de 0.349 * 102
        int coef_g_g = 70; // Aproximación de 0.686 * 102
        int coef_g_b = 17; // Aproximación de 0.168 * 102

        int coef_b_r = 28; // Aproximación de 0.272 * 102
        int coef_b_g = 54; // Aproximación de 0.534 * 102
        int coef_b_b = 13; // Aproximación de 0.131 * 102

        // Aplicar filtro sepia con aproximaciones y desplazamientos
        new_r = (coef_r_r * r + coef_r_g * g + coef_r_b * b) >> 7;
        new_g = (coef_g_r * r + coef_g_g * g + coef_g_b * b) >> 7;
        new_b = (coef_b_r * r + coef_b_g * g + coef_b_b * b) >> 7;

        // Combinar los componentes en un solo p迷el
        curr_pixel.data = (new_b, new_g, new_r);

        stream_out.write(curr_pixel);
    }

}
```

5.1.1.3 Sobel

El filtro Sobel se emplea para detectar los bordes de una imagen mediante el cálculo de gradientes. Se implementó un algoritmo que aplica un kernel Sobel a la imagen para resaltar estos bordes. La optimización se enfocó en la paralelización y la eficiencia del uso de la memoria, utilizando `#pragma HLS unroll` para desenrollar bucles y reducir el número de ciclos necesarios para el procesamiento.

Además, se utiliza la directiva `#pragma HLS PIPELINE II=1` que instruye al compilador HLS para aplicar la técnica de pipeline a un bucle o bloque de código específico, donde `II=1` establece un Intervalo de Inicio de un ciclo de reloj. Esto significa que el sistema puede iniciar una nueva iteración del bucle en cada ciclo de reloj, optimizando así el rendimiento al permitir que varias operaciones se solapen en tiempo. Aunque esta estrategia busca maximizar la eficiencia y el rendimiento del procesamiento, también puede aumentar el uso de recursos del hardware y su viabilidad depende de la complejidad de las dependencias de datos y las capacidades del hardware.

Otro recurso utilizado para mejorar el filtro es la directiva `#pragma HLS inline`. Al aplicar esta directiva a una función, se instruye al compilador para que inserte el cuerpo completo de la función en cada punto donde se la llama, en lugar de manejarla como una llamada de función separada. Esto puede reducir la sobrecarga de las llamadas a funciones y potencialmente aumentar el rendimiento, ya que elimina la necesidad de saltos adicionales y manejo de contexto durante la ejecución. Sin embargo, el uso excesivo de inlining puede llevar a un aumento en el uso de recursos de hardware, ya que el código de la función se duplica en múltiples lugares. La directiva se usa típicamente para optimizar piezas críticas de código donde el impacto de la llamada a funciones es significativo en términos de latencia o uso de recursos.

Código de cabecera HLS:

```
#ifndef _SOBEL_HEADER
#define _SOBEL_HEADER

#include "ap_int.h"
#include "hls_math.h"
#include "hls_stream.h"

struct DATA_PACK {
	ap_uint<8> data;
	ap_uint<1> last;
};
typedef hls::stream<DATA_PACK> AXIS_T;

#define SIZE 3
#define MAX_WIDTH 1920
#define MAX_HIGHT 1080

void sobel(AXIS_T& input, AXIS_T& output, int rows, int cols);

#endif
```

Código del programa principal HLS:

```
#include "sobel.h"

uint8_t apply_sobel(uint8_t img[SIZE][SIZE], int8_t filter[2][SIZE][SIZE]){
#pragma HLS inline
	int8_t Gx=0,Gy=0;
	for(int i=0 ; i<SIZE ; i++){
		#pragma HLS UNROLL
		for(int j=0 ; j<SIZE ; j++){
			#pragma HLS UNROLL
			Gx += img[i][j]*filter[0][i][j];
			Gy += img[i][j]*filter[1][i][j];
		}
	}
	return (uint8_t)hls::sqrt((double)(Gx*Gx+Gy*Gy));
}

void sobel(AXIS_T &input, AXIS_T &output, int rows, int cols){

	#pragma HLS INTERFACE axis register both port=input
	#pragma HLS INTERFACE axis register both port=output
	#pragma HLS INTERFACE s_axilite port=rows bundle=CONTROL_BUS
	#pragma HLS INTERFACE s_axilite port=cols bundle=CONTROL_BUS
	#pragma HLS INTERFACE s_axilite port=return bundle=CONTROL_BUS

	uint8_t LINE_BUFFER[2][MAX_WIDTH], WINDOW_BUFFER[SIZE][SIZE];
	int8_t FILTER[2][SIZE][SIZE]={{{1,0,-1},
	                               {2,0,-2},
	                               {1,0,-1}},
	                              {{1,2,1},
	                               {0,0,0},
	                               {-1,-2,-1}}};

	//input--, output--;
	for(int row=0 ; row<rows ; row++){
		for(int col=0 ; col<cols ; col++){
			#pragma HLS PIPELINE II=1

			for(int k=0 ; k<SIZE ; k++){
				#pragma HLS unroll
				WINDOW_BUFFER[k][0] = WINDOW_BUFFER[k][1];
				WINDOW_BUFFER[k][1] = WINDOW_BUFFER[k][2];
			}
			WINDOW_BUFFER[0][2] = LINE_BUFFER[0][col];
			WINDOW_BUFFER[1][2] = (LINE_BUFFER[0][col] =
LINE_BUFFER[1][col]);
			WINDOW_BUFFER[2][2] = (LINE_BUFFER[1][col] =
input.read().data);

			// bilateral filter
			DATA_PACK datapack;
			datapack.data = apply_sobel(WINDOW_BUFFER,FILTER);
			datapack.last = (row==rows-1 && col==cols-1);
			output.write(datapack);
		}
	}
}
```

5.1.2 Validación y pruebas de funcionamiento

Para cada filtro desarrollado, se realizaron pruebas de validación en Vitis HLS para verificar su correcto funcionamiento. Se emplearon imágenes de prueba para asegurar que los filtros se aplicaban correctamente y que los resultados eran los esperados. La validación se centró en verificar la fidelidad del efecto de cada filtro y en asegurar que no hubiera errores en el procesamiento de los píxeles.

A continuación, se presentan los códigos de los test bench utilizados para poder implementar estas IPs:

Test bench del filtro Negativo:

```
#include "negative_filter.hpp"
#include <iostream>

int main() {
    video_stream in, out;
    pixel_data curr_pixel;

    // Simulación de un píxel en formato RGB (120, 50, 200)
    curr_pixel.data = 0x7850C8;
    in.write(curr_pixel);
    rgb2negative(in, out);
    out.read(curr_pixel);
    std::cout << "Resultado de la conversión a Negativo: " << std::hex <<
curr_pixel.data << std::dec << std::endl;

    return 0;
}
```

Test bench del filtro Sepia:

```
#include "sepia_filter.hpp"
#include <iostream>

int main() {
    video_stream in, out;
    pixel_data curr_pixel;

    // Simulación de un píxel en formato RGB (120, 50, 200)
    curr_pixel.data = 0x7850C8;
    in.write(curr_pixel);
    rgb2sepia(in, out);
    out.read(curr_pixel);
    std::cout << "Resultado de la conversión a Sepia: " << std::hex <<
curr_pixel.data << std::dec << std::endl;

    return 0;
}
```

Test bench del filtro Sobel:

```
#include <opencv2/opencv.hpp>
#include "sobel.h"

AXIS_T input;//[MAX_HIGHT*MAX_WIDTH];
AXIS_T output;//[MAX_HIGHT*MAX_WIDTH];

int main(int argc,char *argv[]){

	cv::Mat srcImg = cv::imread("ex.png",CV_8UC1); int h=srcImg.rows,
w=srcImg.cols;
	cv::Mat outImg = cv::Mat(h,w,CV_8UC1);

	for(int i=0 ; i < h*w ; i++){
		DATA_PACK datapack;
		datapack.data = srcImg.at<uchar>( i/w , i%w );
		datapack.last = (i==h*w-1);
		input.write(datapack);
	}

	std::cout<<"start sobel"<<std::endl;
	sobel(input, output, h, w);
	std::cout<<"end sobel"<<std::endl;

	for(int i=0 ; i < h*w ; i++)
		outImg.data[i] = output.read().data;
	cv::imwrite("./out_sobel.png",outImg);
	cv::imwrite("./out.png",srcImg);

}
```

5.2 Integración de IP en diseño Vivado

Una vez desarrollados y optimizados los filtros de vídeo en Vitis HLS, el siguiente paso en el desarrollo de sistemas de procesamiento de vídeo en hardware implica la integración de estos módulos IP personalizados en un diseño global utilizando Vivado.

5.2.1 Proceso de integración y desafíos encontrados

La integración de las IP de filtros personalizados en el diseño Vivado consistió en varios pasos clave:

1. Exportación de IP desde Vitis HLS: Una vez optimizados y validados los filtros, se generaron los correspondientes archivos de IP para cada filtro (Negativo, Sepia, Sobel) en Vitis HLS. Esto incluyó la generación de archivos de diseño (`.v`), archivos de simulación y archivos de síntesis, junto con un descriptor de IP (`.xml`) que facilita la integración en Vivado.

2. Incorporación al Diseño de Vivado: Las IPs exportadas se añadieron al proyecto de Vivado utilizando el IP Integrator. Se configuraron para que interactuaran adecuadamente con los demás componentes del diseño,

incluido el módulo de salida HDMI, asegurando la correcta transferencia de datos de vídeo entre las IP y el resto del sistema.

3. Conexiones y Validación: Se establecieron las conexiones necesarias entre las IP personalizadas y los interfaces de entrada/salida del SoC Pynq-Z2, prestando especial atención a la sincronización de señales y la gestión de la memoria. La validación del diseño integrado incluyó la revisión de las conexiones y la simulación del diseño para detectar posibles errores o problemas de rendimiento.

Durante el proceso de integración, se encontraron diversos desafíos, entre ellos la sincronización de señales, para asegurar la correcta sincronización entre las señales de entrada/salida de las IP y los componentes de hardware existentes fue esencial para evitar latencias o pérdidas de datos.

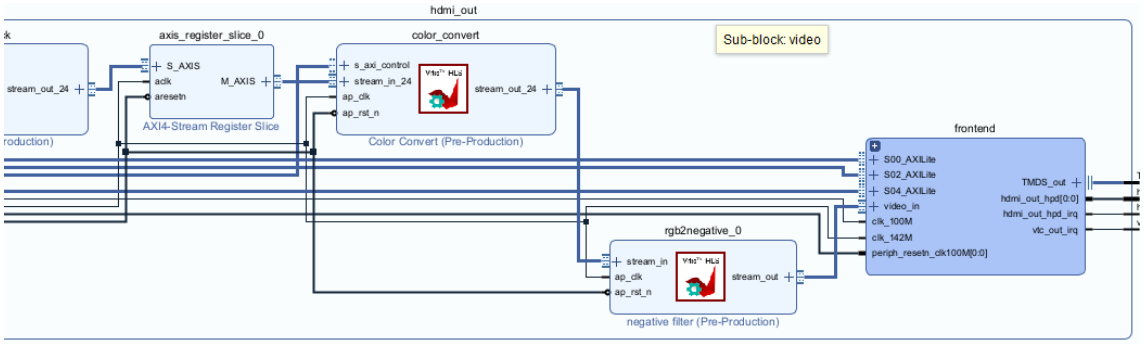

Figura 21 – Integración de la IP del filtro Negativo

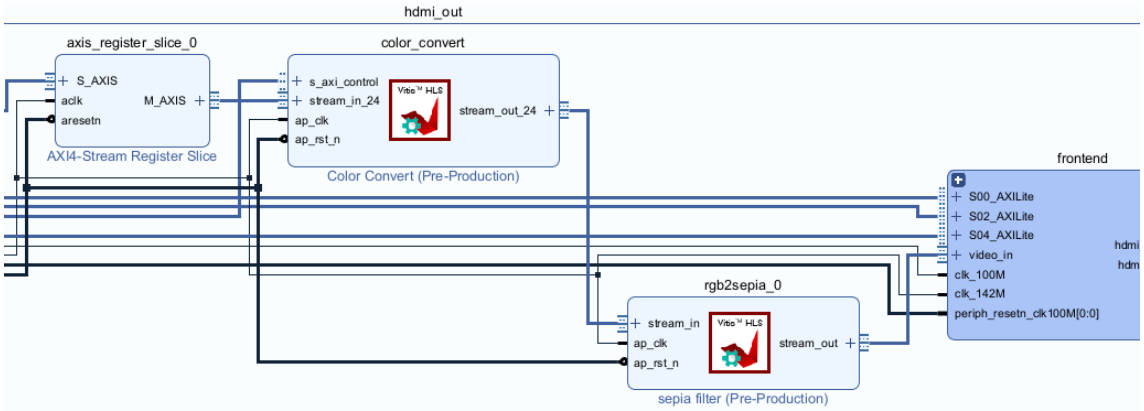

Figura 22 – Integración de la IP del filtro Sepia

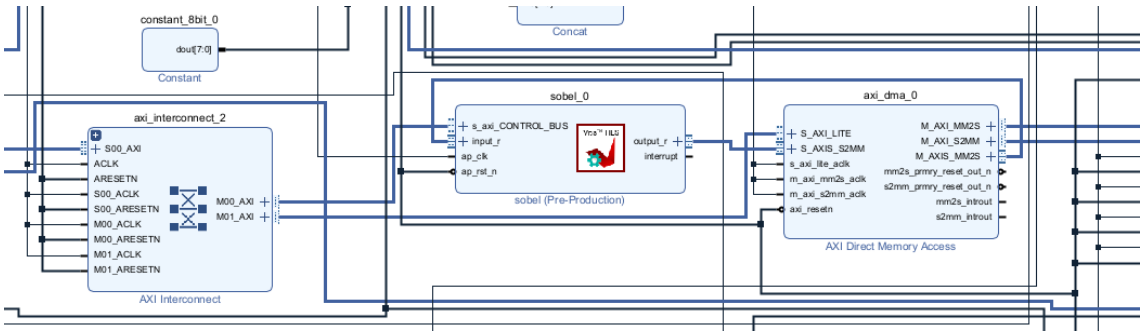

Figura 23 – Integración de la IP del filtro Sobel

Como se puede observar en la Figura 23, el filtro Sobel se ha integrado en el overlay original de la PYNQ-Z2 de forma distinta a los filtros negativo y sepia. En este caso se ha colocado la IP generada en Vitis HLS para el filtro sobel justo antes del acceso directo a memoria (DMA). Esta es otra manera de acelerar el rendimiento, hay muchos métodos de conseguir esta mejora, el único impedimento es la complejidad y compatibilidad de la arquitectura que se implementa.

Es importante recalcar que para la realización del filtro Sobel se ha tenido que actualizar las IPs que Vivado nos proporciona por defecto, ya que algunas de ellas han sido modificadas o incluso eliminadas en las últimas versiones, haciendo incompatible este diseño, por ello si se quiere recrear el overlay se recomienda utilizar los archivos .tcl que se proporcionan junto con este proyecto, archivos que he modificado a mano para así recrear el overlay sin problemas de compatibilidad.

5.2.2 Pruebas de validación y rendimiento

Una vez completada la integración de las IP en Vivado, se llevaron a cabo pruebas exhaustivas para validar la funcionalidad y el rendimiento del diseño integrado. Esto incluyó:

- Pruebas Funcionales: Verificación de que cada filtro procesaba el vídeo de entrada correctamente, generando el efecto deseado sin errores.

- Evaluación del Rendimiento: Medición del tiempo de procesamiento, la latencia y el uso de recursos del FPGA, comparando estos resultados con las expectativas y objetivos del diseño inicial.

Las pruebas de validación y rendimiento confirmaron la exitosa integración de las IP de filtros de vídeo en el hardware, allanando el camino para la implementación final y las pruebas en el entorno de Jupyter.

Además, un desafío afrontar fue la optimización de recursos, es decir, el ajuste de las IP para optimizar el uso de los recursos del FPGA, sin comprometer el

rendimiento del sistema, lo que requirió iteraciones adicionales y pruebas exhaustivas, siendo un proceso bastante largo debido a que por la complejidad del overlay completo de la PYNQ-Z2, a la hora de realizar la parte de simulación, implementación y posterior generación del `.bitstream`, se tarda aproximadamente 2 horas en completar todo el proceso.

5.3 Implementación y pruebas en Jupyter

La última fase del proceso de pruebas hardware implica la implementación efectiva y evaluación de los filtros de vídeo desarrollados en Vitis HLS e integrados en el diseño Vivado, mediante el uso de notebooks de Jupyter en la plataforma Pynq-Z2. Este entorno permite a los usuarios interactuar directamente con el hardware FPGA, cargando el archivo `.bitstream` generado y aplicando los filtros a vídeos en tiempo real.

5.3.1 Descripción del flujo de trabajo

El flujo de trabajo para la implementación y pruebas en Jupyter consta de los siguientes pasos:

1. Carga del `.bitstream`: El primer paso involucra cargar el archivo `.bitstream` correspondiente al diseño Vivado que incluye las IP de los filtros. Esto se realiza utilizando las capacidades de Python disponibles en el entorno Jupyter, específicamente a través de la biblioteca PYNQ que permite la interacción con el hardware de la placa.

2. Aplicación de Filtros: Una vez cargado el diseño en el FPGA, se procede a aplicar los filtros desarrollados a los vídeos de entrada. Esta etapa implica la selección de vídeos de prueba, su procesamiento en tiempo real mediante los filtros de Negativo, Sepia y Sobel, y la visualización de los resultados directamente en el notebook.

3. Evaluación de Resultados: Los vídeos procesados se evalúan en términos de calidad visual, tiempo de procesamiento y eficiencia en el uso de recursos del FPGA. Esta evaluación permite comparar directamente las implementaciones en hardware con las contrapartes en software.

A continuación, se proporcionan los códigos utilizados en cada filtro para este proceso:

Código de la parte hardware del archivo Test-Negativo.ipyb:

```
'''
Hardware
'''
from pynq.overlays.base import BaseOverlay
from pynq.lib.video import *
import cv2
import os
import time
import psutil

# Función para medir la CPU y la memoria utilizada
def measure_resources():
    cpu_percent = psutil.cpu_percent()
    memory_percent = psutil.virtual_memory().percent
    return cpu_percent, memory_percent

# Cargar el overlay
base = BaseOverlay("base_wrapper.bit")

# Configurar la salida HDMI
Mode = VideoMode(640, 480, 24)  # Modo del monitor: 640x480 @ 60Hz
hdmi_out = base.video.hdmi_out
hdmi_out.configure(Mode, PIXEL_BGR)
hdmi_out.start()

# Configuración de la cámara (entrada)
frame_in_w, frame_in_h = 640, 480

# Establecer nivel de registro de OpenCV en silencioso
os.environ["OPENCV_LOG_LEVEL"] = "SILENT"

# Liberar cualquier recurso de cámara existente
cv2.VideoCapture(0).release()

# Inicializar la cámara
videoIn = cv2.VideoCapture(0)
videoIn.set(cv2.CAP_PROP_FRAME_WIDTH, frame_in_w)
videoIn.set(cv2.CAP_PROP_FRAME_HEIGHT, frame_in_h)

# Inicialización de variables para el cálculo de frames por segundo
start_time = time.time()
num_frames = 0
read_errors = 0
total_cpu_usage = 0
total_memory_usage = 0

while (base.buttons[3].read()==0):
    fstime=time.time()

    # Capturar video frame por frame
    ret, frame_vga = videoIn.read()

    if ret:
        # Incrementar el contador de frames
        num_frames += 1

        # Aquí se aplica el procesamiento de imagen si es necesario
        processed_frame = frame_vga

        # Mostrar el frame procesado a través de HDMI Out
        outframe = hdmi_out.newframe()
        outframe[0:frame_in_h, 0:frame_in_w, :] = processed_frame
        cv2.putText(outframe,"FPS:"+str(round(1/(time.time()-
fstime),4)),(50,25),0,0.8,(255,255,255),1)
        hdmi_out.writeframe(outframe)
```

```
        # Medir recursos después de aplicar el filtro
        cpu_usage, memory_usage = measure_resources()
        total_cpu_usage += cpu_usage
        total_memory_usage += memory_usage
    else:
        # Incrementar el contador de errores de lectura
        read_errors += 1
        print("Error al leer de la cámara.")

# Calcular el tiempo transcurrido
end_time = time.time()
elapsed_time = end_time - start_time

# Calcular los frames por segundo
fps = (num_frames - read_errors) / elapsed_time

# Calcular el promedio de uso de CPU y memoria
avg_cpu_usage = total_cpu_usage / num_frames
avg_memory_usage = total_memory_usage / num_frames

# Mostrar resultados
print("Programa finalizado")
print("Frames por segundo: " + str(fps))
print("Número de errores leídos: " + str(read_errors))
print("Uso promedio de CPU: " + str(avg_cpu_usage) + "%")
print("Uso promedio de memoria: " + str(avg_memory_usage) + "%")

# Liberar recursos
videoIn.release()
hdmi_out.stop()
hdmi_out.close()
```

Código de la parte hardware del archivo Test-Sepia.ipyb:

```
'''
Hardware
'''
from pynq.overlays.base import BaseOverlay
from pynq.lib.video import *
from pynq import Overlay
import cv2
import os
import time
import psutil

# Función para medir la CPU y la memoria utilizada
def measure_resources():
    cpu_percent = psutil.cpu_percent()
    memory_percent = psutil.virtual_memory().percent
    return cpu_percent, memory_percent

# Cargar el overlay
base = BaseOverlay("base_wrapper.bit")

# Configurar la salida HDMI
Mode = VideoMode(640, 480, 24)  # Modo del monitor: 640x480 @ 60Hz
hdmi_out = base.video.hdmi_out
hdmi_out.configure(Mode, PIXEL_BGR)
hdmi_out.start()

# Configuración de la cámara (entrada)
frame_in_w, frame_in_h = 640, 480
```

```
# Establecer nivel de registro de OpenCV en silencioso
os.environ["OPENCV_LOG_LEVEL"] = "SILENT"

# Liberar cualquier recurso de cámara existente
cv2.VideoCapture(0).release()

# Inicializar la cámara
videoIn = cv2.VideoCapture(0)
videoIn.set(cv2.CAP_PROP_FRAME_WIDTH, frame_in_w)
videoIn.set(cv2.CAP_PROP_FRAME_HEIGHT, frame_in_h)

# Inicialización de variables para el cálculo de recursos
start_time = time.time()
num_frames = 0
read_errors = 0
total_cpu_usage = 0
total_memory_usage = 0

while (base.buttons[3].read()==0):
    # Capturar video frame por frame
    fstime=time.time()
    ret, frame_vga = videoIn.read()

    if ret:
        # Incrementar el contador de frames
        num_frames += 1

        # Aquí se aplica el procesamiento de imagen si es necesario
        processed_frame = frame_vga

        # Mostrar el frame procesado a través de HDMI Out
        outframe = hdmi_out.newframe()
        outframe[0:frame_in_h, 0:frame_in_w, :] = processed_frame
        cv2.putText(outframe,"FPS:"+str(round(1/(time.time()-
fstime),4)),(50,25),0,0.8,(255,255,255),1)
        hdmi_out.writeframe(outframe)

        # Medir recursos después de aplicar el filtro
        cpu_usage, memory_usage = measure_resources()
        total_cpu_usage += cpu_usage
        total_memory_usage += memory_usage
    else:
        # Incrementar el contador de errores de lectura
        read_errors += 1
        print("Error al leer de la cámara.")

# Calcular el tiempo transcurrido
end_time = time.time()
elapsed_time = end_time - start_time

# Calcular el promedio de uso de CPU y memoria
avg_cpu_usage = total_cpu_usage / num_frames
avg_memory_usage = total_memory_usage / num_frames

# Calcular los frames por segundo (FPS)
fps = num_frames / elapsed_time

# Mostrar resultados
print("Programa finalizado")
print("Frames por segundo: " + str(fps))
print("Número de errores leídos: " + str(read_errors))
print("Uso promedio de CPU: " + str(avg_cpu_usage) + "%")
print("Uso promedio de memoria: " + str(avg_memory_usage) + "%")

# Liberar recursos
videoIn.release()
hdmi_out.stop()
hdmi_out.close()
```

Código de la parte hardware del archivo Test-USB2HDMI_Sobel.ipyb:

```
import cv2, time, warnings
import numpy as np
from pynq.lib.video import *
warnings.filterwarnings('ignore')
import pynq
from pynq.overlays.base import BaseOverlay
from pynq import Overlay
from pynq import allocate
import psutil

# Función para medir la CPU y la memoria utilizada
def measure_resources():
    cpu_percent = psutil.cpu_percent()
    memory_percent = psutil.virtual_memory().percent
    return cpu_percent, memory_percent

overlay = BaseOverlay('base_w_sobel.bit')

sobel = overlay.sobel_0
dma = overlay.axi_dma_0

def reset_all_dma():
    def dma_reset(dma):
        dma.sendchannel.stop()
        dma.recvchannel.stop()
        dma.sendchannel.start()
        dma.recvchannel.start()
    dma_reset(dma)
reset_all_dma()

resolution=(640,480)
padd=int((max(resolution)-min(resolution))/2)
recive_head=0+padd
recive_tail=recive_head+min(resolution)
buffer_shape=max(resolution)

input_buffer = allocate(shape=(buffer_shape,buffer_shape), dtype=np.uint8)
output_buffer = allocate(shape=(buffer_shape,buffer_shape), dtype=np.uint8)

def run_sobel():
    reset_all_dma()
    dma.sendchannel.transfer(input_buffer)
    dma.recvchannel.transfer(output_buffer)
    sobel.write(0x10,buffer_shape)
    sobel.write(0x18,buffer_shape)
    sobel.write(0x00,0x81) #start
    dma.sendchannel.wait()
    dma.recvchannel.wait()

frame_in_w,frame_in_h = resolution
cap = cv2.VideoCapture(0)
cap.set(cv2.CAP_PROP_FRAME_WIDTH, frame_in_w);
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, frame_in_h);
cap.isOpened()

'''
Con acelerador PL
'''
frame_out_w,frame_out_h = resolution
Mode = VideoMode(frame_out_w,frame_out_h,8)
hdmi_out = overlay.video.hdmi_out
hdmi_out.configure(Mode,PIXEL_GRAY)
hdmi_out.start()

start=time.time()
```

```
num_frames=0
readError=0
total_cpu_usage = 0
total_memory_usage = 0
while (overlay.buttons[3].read()==0):
    fstime=time.time()
    frame, image=cap.read()
    if (frame):
        #image=cv2.GaussianBlur(image,(3,3),0)
        image=cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
        image=
cv2.copyMakeBorder(image,padd,padd,0,0,cv2.BORDER_CONSTANT,value=[0,0,0])
        input_buffer[:,:] = image
        run_sobel()
        outframe=hdmi_out.newframe()
        outframe=output_buffer[recive_head:recive_tail,:]
        cv2.putText(outframe,"FPS:"+str(round(1/(time.time()-
fstime),4)),(1,20),0,0.8,(255,255,255),1)
        hdmi_out.writeframe(outframe)
        num_frames+=1
        # Medir recursos después de procesar el frame
        cpu_usage, memory_usage = measure_resources()
        total_cpu_usage += cpu_usage
        total_memory_usage += memory_usage
    else:
        readError+=1
        print("Error al leer de la cámara.")

end=time.time()
print("Prueba finalizada")
print("Frames por segundo: " + str((num_frames-readError) / (end - start)))
print("Uso promedio de CPU: " + str(total_cpu_usage / num_frames) + "%")
print("Uso promedio de Memoria: " + str(total_memory_usage / num_frames) +
"%")
print("Número de errores leídos: " + str(readError))
```

5.3.2 Análisis de resultados y eficiencia

En este apartado se analizan los resultados obtenidos de las pruebas y evaluaciones realizadas sobre los sistemas SoC en el contexto del procesado de vídeo mediante hardware, utilizando para ello las mismas funciones especificadas anteriormente de la librería `psutil`. Este análisis detallado proporciona una comprensión más profunda de la eficacia y eficiencia de las implementaciones, además de ilustrar las capacidades reales de estos sistemas en aplicaciones prácticas de procesamiento en tiempo real. Más adelante, en el capítulo 6, discutiremos los hallazgos clave que destacan tanto las ventajas como las limitaciones de los sistemas SoC comparados con arquitecturas de hardware tradicionales, enfatizando su potencial para transformar significativamente el procesamiento de vídeo en diversos entornos industriales y comerciales.

N.º de Intento	**1**	**2**	**3**	**4**	**5**
Frames por segundo	10.5091	12.0765	11.7824	11.9439	11.1309
Errores leídos	0	0	0	0	0
Uso promedio de CPU	20.7639%	21.5402%	23.7893%	21.4548%	31.2397%
Uso promedio de memoria	60.9443%	60.9444%	47.8770%	47.8962%	57.6414%

Tabla 4 – Resultados de aplicar un filtro negativo en hardware

N.º de Intento	1	2	3	4	5
Frames por segundo	11.9620	11.2250	10.9907	12.1975	12.1287
Errores leídos	0	0	0	0	0
Uso promedio de CPU	20.8350%	22.8567%	29.0663%	23.7025%	21.7362%
Uso promedio de memoria	67.1427%	69.3617%	63.6842%	62.6719%	62.6114%

Tabla 5 – Resultados de aplicar un filtro sepia en hardware

N.º de Intento	1	2	3	4	5
Frames por segundo	12.0500	12.2450	12.2901	12.2499	12.1085
Errores leídos	0	0	0	0	0
Uso promedio de CPU	34.9812%	33.7215%	29.7340%	26.7134%	27.4336%
Uso promedio de memoria	46.8103%	46.9318%	46.9185%	46.8773%	47.1664%

Tabla 6 – Resultados de aplicar un filtro sobel en hardware

5.3.2.1 Conclusiones

La implementación y pruebas de filtros de vídeo en hardware utilizando el SoC Pynq-Z2 demuestra las capacidades avanzadas de procesamiento que ofrece el hardware FPGA frente a soluciones puramente basadas en software. A través de este proceso, desde el desarrollo de filtros en Vitis HLS hasta su evaluación en Jupyter, se evidencian las ventajas en términos de rendimiento, eficiencia y calidad que hacen del procesamiento en hardware una opción atractiva para aplicaciones de vídeo en tiempo real.

6 COMPARACIÓN Y ANÁLISIS DE PRESTACIONES

El análisis de prestaciones es fundamental para entender el impacto de las decisiones de diseño en el procesamiento de vídeo, particularmente cuando se comparan implementaciones en software y hardware. En este capítulo se va a presentar una comparación y análisis de las distintas implementaciones realizadas.

6.1 Evaluación de tiempos de procesamiento

6.1.1 Metodología para medir los tiempos de procesamiento

Para evaluar los tiempos de procesamiento de manera efectiva, se debe establecer una metodología clara y consistente que permita mediciones precisas y comparables. Los siguientes pasos describen un enfoque típico.

La metodología para medir los tiempos de procesamiento en este estudio se centró en garantizar precisión y reproducibilidad. Se implementaron las siguientes etapas:

- **Definición de los escenarios de prueba**: Los escenarios de prueba se diseñaron para emular condiciones reales de uso, abarcando diferentes tipos de contenido de vídeo. Se seleccionaron vídeos de varias duraciones y complejidades para evaluar cómo los filtros afectan el rendimiento en situaciones diversas.

- **Entorno de pruebas controlado**: Para minimizar las variaciones externas, se estableció un entorno de pruebas controlado. Esto implicó el uso de una plataforma de hardware y software constante, con los mismos recursos de sistema disponibles para cada prueba, asegurando que las comparaciones entre software y hardware fueran justas y consistentes, tratando que el código final de Jupyter fuera lo más similar posible entre software y hardware.

- **Medición de tiempos**: La medición de tiempos se realizó utilizando herramientas de software especializadas, concretamente las librerías de Python `time` y `psutil`, librerías, capaces de registrar con precisión el tiempo total de procesamiento desde la carga del vídeo hasta la finalización del filtro. Estas mediciones se llevaron a cabo múltiples veces para cada configuración de prueba para asegurar la fiabilidad de los datos.

- **Repetición de pruebas**: Cada escenario de prueba se repitió varias veces (típicamente cinco o más) para garantizar la consistencia de los resultados. La repetición permitió identificar y mitigar anomalías, proporcionando un conjunto de datos más robusto para el análisis.

6.1.2 Análisis comparativo entre software y hardware

Una vez recogidos y analizados los datos, el siguiente paso es realizar un análisis comparativo del rendimiento de las implementaciones en software frente a las de hardware, considerando los tiempos de procesamiento como la métrica principal.

La comparación entre las implementaciones de software y hardware de los filtros de procesado de vídeo (negativo, sepia, y Sobel) se fundamenta en varios criterios clave: rendimiento, eficiencia, impacto en la experiencia del usuario y costo-beneficio.

6.1.2.1 Comparación de rendimiento

La comparación de rendimiento se basa en los fotogramas por segundo (FPS) alcanzados durante las pruebas. Los resultados muestran que la implementación en hardware supera en los filtros de mayor complejidad a los diseñados puramente en software, sin embargo, el filtro negativo tiene mayor rendimiento en software con respecto al diseño adoptado en hardware. Esta medida indica una mayor fluidez y capacidad de procesamiento en tiempo real con forme más frames (fotogramas) por segundo. Las Tablas 7, 8 y 9 muestran los resultados de frames por segundo para los filtros Negativo, Sepia y Sobel en sus versiones software y hardware.

N.º de Intento	1	2	3	4	5	Media
Software	11.8927	12.2085	12.2170	11.8929	11.9718	12.0366
Hardware	10.5091	12.0765	11.7824	11.9439	11.1309	11.4886

Tabla 7 – Frames Por Segundo del filtro Negativo

N.º de Intento	1	2	3	4	5	Media
Software	8.9993	9.1438	9.2927	9.2873	9.3750	9.2196
Hardware	11.9620	11.2250	10.9907	12.1975	12.1287	11.7008

Tabla 8 – Frames Por Segundo del filtro Sepia

N.º de Intento	1	2	3	4	5	Media
Software	9.6581	9.7458	9.4564	9.8303	9.4513	9.6284
Hardware	12.0500	12.2450	12.2901	12.2499	12.1085	12.1887

Tabla 9 – Frames Por Segundo del filtro Sobel

6.1.2.2 Análisis de eficiencia

La eficiencia se evaluó en términos de consumo de CPU y memoria. Para ello, primero se ha configurado el proyecto Jupyter, y se han medido estos parámetros en el momento en el que se empieza a aplicar el filtro, buscando ser lo más objetivo y similares posibles.

Primero se muestran los resultados del uso promedio de CPU, medida fundamental en la capacidad de procesamiento de una placa. En la Tabla 10 puede observarse cómo este uso de CPU es significativamente inferior en hardware al que se utiliza en software, por lo que optar por la implementación adoptada para el filtro negativo es una solución muy eficiente en cuanto a términos de uso de CPU.

N.º de Intento	1	2	3	4	5	Media
Software	26.6382%	33.6833%	29.3671%	28.9915%	32.2249%	30.1810%
Hardware	20.7639%	21.5402%	23.7893%	21.4548%	31.2397%	23.7576%

Tabla 10 – Uso promedio de CPU para el filtro Negativo

Para el filtro Sepia (Tabla 11) se obtienen resultados similares, ya que el uso de CPU es muchísimo menor en hardware al que se usa en software. Es conveniente recordar que el filtro hardware se implementó utilizando aproximaciones ya que no cumplía con las restricciones impuestas para el dispositivo en Vitis HLS. Aun así, los resultados visuales son muy similares, y el rendimiento en cuanto a CPU es de aproximadamente el doble para el filtro desarrollado en hardware a diferencia del desarrollado en software, incrementando el uso de CPU.

N.º de Intento	1	2	3	4	5	Media
Software	61.3549%	56.5452%	56.3512%	56.3890%	56.8657%	57.5012%
Hardware	20.8350%	22.8567%	29.0663%	23.7025%	21.7362%	23.6393%

Tabla 11 – Uso promedio de CPU para el filtro Sepia

Por último, la Tabla 12 muestra los resultados obtenidos para el filtro Sobel tanto en software como en hardware. Este filtro se ha interconectado en Vivado con las IPs de la PYNQ-Z2 de una manera distinta a la que se han hecho los filtros anteriores, conectándose aguas abajo del Direct Memory Access. Como resultado, el uso promedio de CPU es de aproximadamente un 27% menor en hardware al utilizado en software.

N.º de Intento	1	2	3	4	5	Media
Software	56.0400%	57.0178%	55.1263%	61.1178%	56.1521%	57.0908%
Hardware	34.9812%	33.7215%	29.7340%	26.7134%	27.4336%	30.5167%

Tabla 12 – Uso promedio de CPU para el filtro Sobel

En cuanto al uso promedio de memoria para el filtro Negativo (Tabla 13), es más eficiente en términos de memoria usar la implementación hardware, aproximadamente un 3.5% más eficiente en hardware.

N.º de Intento	1	2	3	4	5	Media
Software	59.3131%	58.1083%	60.7842%	54.0340%	60.8463%	58.6172%
Hardware	60.9443%	60.9444%	47.8770%	47.8962%	57.6414%	55.0607%

Tabla 13 – Uso promedio de memoria para el filtro Negativo

El uso de memoria promedio para el filtro sepia (Tabla 14) es, a diferencia que su utilización de CPU, menor en el diseño software al hardware, aunque en ambos casos son muy similares. Para este tipo de situaciones es conveniente plantearse si merece la pena realizar un diseño hardware (más complejo que el software) ante tan poca mejora, o incluso como en ese caso, siendo un poco peor en cuanto a uso de memoria.

N.º de Intento	1	2	3	4	5	Media
Software	62.6431%	62.6426%	62.6856%	63.6625%	63.5090%	63.0286%
Hardware	67.1427%	69.3617%	63.6842%	62.6719%	62.6114%	65.0944

Tabla 14 – Uso promedio de memoria para el filtro Sepia

Por último, tenemos el caso del filtro Sobel (Tabla 15), en el cual ambos diseños utilizan un gran porcentaje de memoria debido a los requerimientos computacionales de implementar este filtro de detección de bordes. Aunque ambos usos promedios de memoria son relativamente similares, la implementación hardware es nuevamente más eficiente para este filtro.

N.º de Intento	1	2	3	4	5	Media
Software	49.6552%	49.6800%	57.8158%	47.6122%	47.4275%	50.4381%
Hardware	46.8103%	46.9318%	46.9185%	46.8773%	47.1664%	46.9409%

Tabla 15 – Uso promedio de memoria para el filtro Sobel

6.1.2.3 Impacto en la experiencia del usuario

El rendimiento mejorado y la mayor eficiencia del hardware tienen un impacto positivo en la experiencia del usuario, ofreciendo una reproducción más fluida y tiempos de respuesta más rápidos, factores críticos especialmente en aplicaciones de procesamiento de vídeo en tiempo real. Aunque no menos importante que las anteriores, esta medida es más subjetiva, aunque en general, obviando el filtro Sepia que se realizó en Vitis HLS mediante aproximaciones, en todos los casos se refleja mayor nitidez y calidad en los diseños hardware. Las figuras 24 a 29 así lo muestran.

Figura 24 – Fotografía del vídeo procesado para el filtro negativo mediante software

Figura 25 – Fotografía del vídeo procesado para el filtro negativo mediante hardware

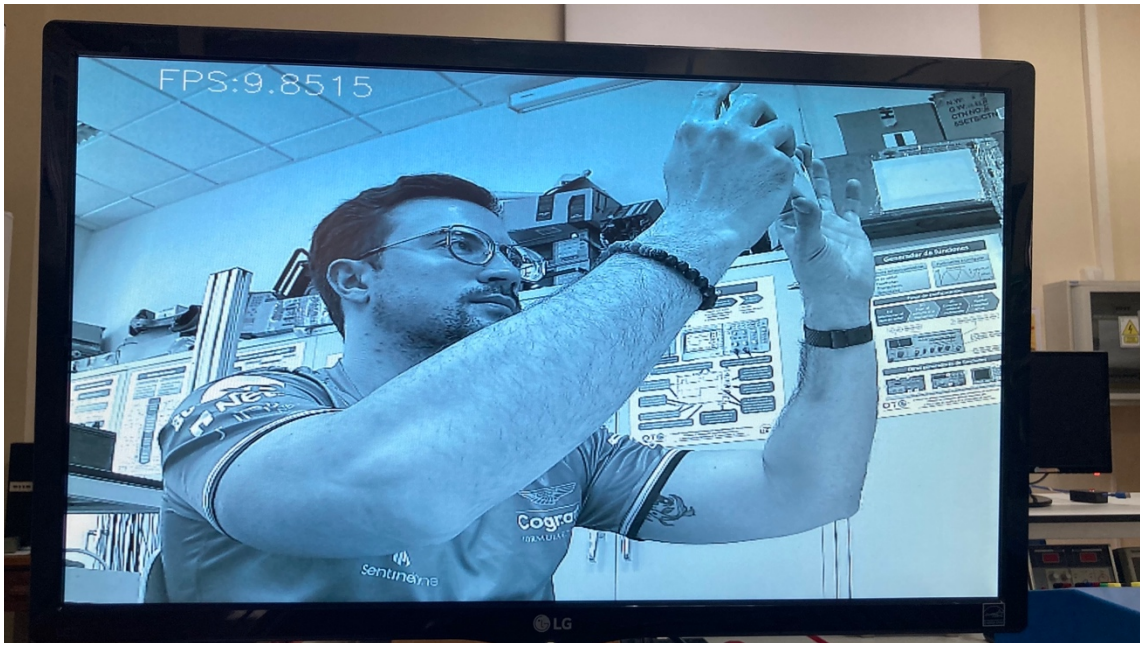

Figura 26 – Fotografía del vídeo procesado para el filtro sepia mediante software

Figura 27 – Fotografía del vídeo procesado para el filtro sepia mediante hardware

Figura 28 – Fotografía del vídeo procesado para el filtro sobel mediante software

Figura 29 – Fotografía del vídeo procesado para el filtro sobel mediante hardware

6.1.2.4 Costo-beneficio

Aunque la implementación en hardware puede conllevar costos iniciales más altos debido a la necesidad de hardware especializado (como el SoC Pynq-Z2), el aumento significativo en el rendimiento y la eficiencia puede justificar la

inversión, especialmente para aplicaciones críticas que requieren un procesamiento de vídeo eficiente y de alta calidad.

Los resultados destacan las ventajas de la implementación en hardware en términos de rendimiento y eficiencia. Aunque el costo inicial y la complejidad de desarrollo pueden ser mayores, los beneficios en términos de capacidad de procesamiento y optimización de recursos hacen que la inversión en hardware sea altamente recomendable para aplicaciones que demanden un alto rendimiento de procesado de vídeo.

6.2 Análisis de uso de recursos

Este análisis se enfoca en cómo las implementaciones de software y hardware de los filtros de procesado de vídeo afectan el consumo de recursos del sistema, incluyendo la CPU y la memoria.

6.2.1 Evaluación del consumo de recursos en ambas implementaciones

Los datos indican que la implementación en hardware generalmente, aunque no siempre, consume menos recursos del sistema en comparación con la implementación en software. Esto se manifiesta en un menor porcentaje de uso de CPU y en un uso más eficiente de la memoria, lo cual es particularmente importante en sistemas con recursos limitados o cuando se ejecutan múltiples tareas simultáneamente. Por lo tanto, a la conclusión que se ha llegado es que dependiendo de los resultados o funciones que se busquen conseguir a la hora de realizar un diseño, optar por una solución u otra (hardware o software) dependiendo de los criterios que se tengan en cuenta.

6.2.2 Impacto en la eficiencia y escalabilidad del sistema

El menor consumo de recursos no solo mejora la eficiencia operativa del sistema actual, sino que también contribuye a una mayor escalabilidad. Al reducir la carga sobre la CPU y optimizar el uso de la memoria, se facilita la incorporación de más funciones de procesamiento de vídeo o la ejecución de tareas adicionales en paralelo sin degradar el rendimiento.

Esta sección resalta la importancia de considerar el uso de recursos al elegir entre implementaciones de software y hardware para el procesamiento de vídeo. Aunque el hardware puede requerir una inversión inicial mayor, su capacidad para manejar el procesamiento de manera más eficiente desde el punto de vista de los recursos puede ofrecer beneficios significativos a largo plazo, especialmente en aplicaciones donde el rendimiento y la eficiencia son críticos. Por el contrario, la aplicación que se quiere realizar puede adaptarse mejor desarrollando su implementación en software en ciertos casos o para los criterios que se tengan en cuenta.

6.3 Comparación de eficiencia entre software y hardware

Esta sección resume los hallazgos clave en términos de rendimiento, uso de recursos, eficiencia y escalabilidad, destacando las diferencias fundamentales entre las implementaciones en software y hardware.

6.3.1 Resumen de hallazgos clave

6.3.1.1 Diferencias en Tiempos de Inicio

Analizamos los tiempos requeridos desde el inicio del procesamiento hasta la primera salida útil. El hardware demostró una inicialización más rápida, fundamental para aplicaciones que demandan respuestas inmediatas.

6.3.1.2 Capacidad de procesamiento en condiciones de alta carga

Se evaluó cómo cada implementación maneja cargas de trabajo incrementadas, especialmente en términos de latencia y estabilidad del sistema. El hardware mantuvo un rendimiento consistente, mientras que el software exhibió una degradación proporcional al aumento de carga, con fallos a la hora de reiniciar el programa, detener y reanudar el bucle que implementa la aplicación, o a la hora de inicializar los recursos a utilizar, como con fallos a la hora de inicializar la cámara, por ejemplo.

También es importante recalcar que si puede mejorar la eficiencia tanto como se quiera, haciendo más simple o complejo el diseño, por ejemplo, se podría cambiar el overlay original de PYNQ-Z2 para procesar los píxeles de dos en dos, en lugar de uno en uno como se hace en el bloque HDMI Out.

6.3.1.3 Flexibilidad en la implementación de nuevos algoritmos

Consideramos la facilidad con la cual se pueden adaptar o implementar nuevos algoritmos de procesamiento de vídeo. Mientras que las modificaciones en hardware requieren un esfuerzo de diseño y síntesis mayor, ofrecen mejoras sustanciales en rendimiento una vez implementadas, además de otro aspecto importante como es la modularidad, es decir, el poder adoptar este algoritmo desarrollado en programas como Vitis HLS para la implementación en Vivado, tanto de la PYNQ-Z2 como de cualquier otro dispositivo de desarrollo.

6.3.2 Implicaciones prácticas y teóricas

La superioridad del hardware en términos de rendimiento y eficiencia tiene implicaciones importantes tanto para el diseño de sistemas de procesamiento de vídeo como para la toma de decisiones en el desarrollo de aplicaciones que requieren un alto grado de procesamiento en tiempo real. La elección entre software y hardware debe considerar no solo el costo y la complejidad de desarrollo sino también los beneficios a largo plazo en términos de rendimiento, eficiencia y escalabilidad.

7 CONCLUSIONES Y TRABAJO FUTURO

El objetivo de este proyecto fue implementar y analizar filtros de procesado de vídeo en sistemas SoC-FPGA. Para conseguirlo ha habido que estudiar las metodologías de diseño y seleccionar una tecnología sobre la que realizar los diseños, comparar distintas las opciones de implementación y el comportamiento en el rendimiento y la eficiencia de las implementaciones hardware frente a las implementaciones de software tradicionales. Se ha logrado demostrar que, a través del diseño cuidadoso en Vitis HLS y la integración en Vivado, es posible alcanzar un procesamiento de vídeo en hardware más rápido y eficiente en cierto tipo de aplicaciones.

Para conseguir este objetivo se han seguido las siguientes etapas y actividades:

1. **Configuración del entorno de desarrollo**: Preparación del entorno de desarrollo necesario para las pruebas y simulaciones. Se seleccionaron los dispositivos de Zynq de Xilinx para realizar la implementación, y en concreto la plataforma SoC Pynq-Z2. Se escogió el software de desarrollo Vitis 2022.2 para la realización de los diseños. Y para la metodología de diseño, se decidió no sólo seguir una metodología a nivel RT (utilizando la herramienta de Vivado), sino también utilizar metodologías basadas en diseño de alto nivel, en este caso utilizando la herramienta Vitis-HLS.

2. **Familiarización con la plataforma Pynq-Z2**: Se realizó un estudio y comprensión de las capacidades y características de la plataforma Pynq-Z2 utilizada para el desarrollo del proyecto. En concreto se profundizó en las alternativas para programar el software en el dispositivo Zynq de esta placa y para comunicar el sistema de procesado software (PS) con la parte de programación FPGA (PL). De las opciones disponibles se escogió usar la utilidad que proporciona la playa Pynq-Z2 de tener integrada una plataforma Jupyter al arrancar con el sistema operativo Linux y así programar el software utilizando Python.

3. **Pruebas iniciales**: Se realizaron una serie de pruebas iniciales para poner a punto la metodología de diseño y asegurar el correcto funcionamiento de los diseños, tanto hardware como software en la plataforma seleccionada. Concretamente se realizaron los siguientes diseños, los cuales son simples pero proporcionan toda la información necesaria para realizar un diseño más complejo como los últimos filtros realizados (negativo, sepia y sobel):

    - Se comenzó con un diseño que consistía en hacer parpadear los LEDs de la Pynq-Z2 en función de los switches que se pulsen, enseñando a controlar LEDs y switches mediante Python y el entorno PYNQ. El enfoque metodológico incluye una explicación teórica de la arquitectura y componentes de la placa, seguida de instrucciones detalladas para configurar el entorno de desarrollo, incluyendo la instalación del software necesario y la preparación de la tarjeta SD.

- El segundo diseño tiene como objetivo principal enseñar cómo utilizar el lenguaje de descripción de hardware Verilog junto con el entorno de desarrollo Xilinx Vivado para programar la placa PYNQ-Z2. El enfoque metodológico incluye una introducción teórica a Verilog y Vivado, seguida de instrucciones detalladas para configurar el entorno de desarrollo, incluyendo la instalación de Vivado y la preparación de los archivos de configuración. El tutorial proporciona ejemplos de código en Verilog, mostrando cómo escribir, simular y sintetizar diseños de hardware, así como cómo programar la FPGA de la PYNQ-Z2. A través de este proceso, se aprende a crear y probar diseños de hardware personalizados, integrando conceptos de diseño digital con la programación de FPGAs.

- El tercer diseño tiene como objetivo enseñar a los usuarios cómo utilizar las fuentes de reloj de la FPGA PYNQ-Z2 para realizar operaciones de temporización. La metodología incluye una introducción a las fuentes de reloj disponibles en la PYNQ-Z2, instrucciones sobre cómo configurar el entorno de desarrollo para trabajar con estos relojes y ejemplos prácticos de cómo dividir la señal de reloj para controlar la velocidad de parpadeo de los LEDs. Los usuarios aprenderán a utilizar módulos de Verilog para manejar la lógica del reloj y crear proyectos prácticos que involucren la manipulación de la frecuencia del reloj de la FPGA.

- Posteriormente, se probó la transmisión de imágenes desde el SoC hacia un monitor externo. Implicando la interacción con buffers de vídeo y la gestión de la señal HDMI. Este proceso permitió una práctica inicial en el procesamiento de vídeo a nivel de hardware y una experiencia tangible en la manipulación de flujos de vídeo. Además, el desarrollo de un script en Jupyter para enviar imágenes por HDMI validó la capacidad del hardware para el procesamiento intensivo de vídeo y demostró su potencial en proyectos avanzados de procesamiento de vídeo.

- El siguiente diseño realizado fue la implementación del diseño "addmul" en Vitis HLS, esencial para entender la colaboración entre la síntesis de alto nivel y la plataforma PYNQ-Z2. Aunque simple, demostró el flujo completo de desarrollo de una IP personalizada, desde la codificación en C/C++ hasta su integración en un sistema embebido. El proceso mostró cómo traducir y optimizar algoritmos para FPGA, y culminó con el uso de esta IP en Python desde Jupyter Notebook. Este proyecto destacó la transición de la conceptualización a la implementación práctica en hardware.

- Durante nuestro estudio del procesamiento de vídeo en hardware con sistemas SoC, adaptamos el diseño del hardware para satisfacer demandas específicas. Esto implicó la generación de módulos personalizados, la integración de IPs en Vivado y el uso de herramientas como Vitis HLS y Jupyter. Un ejemplo clave fue la modificación del bloque HDMI Out para cambiar el formato de imagen de RGB a YUV. Este

proceso incluyó agregar una IP de conversión de color y ajustar el overlay de la PYNQ-Z2, demostrando la flexibilidad y potencial de personalización en proyectos avanzados de procesamiento de vídeo.

4. **Desarrollo de filtros de video implementados en software**: El flujo de diseño incluye la configuración inicial y pruebas del sistema para verificar los resultados, seguida de una fase de evaluación comparativa entre procesamiento de vídeo en software y hardware. Este análisis implica el desarrollo de pruebas específicas en Jupyter Notebook, midiendo tiempo de ejecución, uso de recursos y calidad del vídeo procesado. La metodología se enfoca en la replicabilidad y objetividad, utilizando herramientas como OpenCV y NumPy en un entorno consistente. Los resultados proporcionan una base sólida para evaluar la eficacia y eficiencia de ambos enfoques en proyectos de procesamiento de vídeo. Esta fase busca comprender las ventajas y limitaciones de ambos enfoques y generar datos empíricos sobre rendimiento, eficiencia de recursos y calidad del resultado. Las pruebas en Jupyter Notebook con filtros como negativo, sepia y sobel, medirán el tiempo de ejecución, uso de recursos y calidad del vídeo procesado. La comparación ayudará a tomar decisiones informadas en proyectos de procesamiento de vídeo. Primero se realizaron las pruebas puramente con diseño software, sin modificar el overlay original, es decir, sin modificar el hardware.

5. **Desarrollo de filtros de vídeo en hardware utilizando Vitis HLS**: En un paso posterior, se realizó el diseño y optimización de filtros de vídeo en hardware, partiendo de descripciones a alto nivel y utilizando la herramienta Vitis HLS. Se realizaron las correspondientes pruebas de validación y pruebas de funcionamiento de los filtros diseñados. En concreto se diseñaron en Vitis HLS los mismos filtros mencionados en el apartado anterior (negativo, sepia y sobel) intentando seguir el mismo criterio de diseño y haciéndolos lo más similares posibles, pero esta vez en hardware mediante su descripción en lenguaje C/C++. Una vez diseñados estos filtros se introducen en el HDMI Out de la placa de desarrollo mediante Vivado, para que una vez se le realicen la síntesis e implementación, obtener el overlay final para su posterior prueba y comparación en Jupyter.

6. **Integración de IP en diseño Vivado**: Integración de los filtros diseñados como IP en el entorno de diseño de Vivado, y realización de pruebas de validación y rendimiento para asegurar la correcta funcionalidad del diseño integrado. Para los filtros negativo y sepia bastó con interconectar la IP del “color_covert” del HDMI Out con la IP diseñada y la salida al bloque “frontend”, el cual configura los píxeles para su correcta representación en el monitor. Pero en el caso del filtro sobel se tuvo que hacer una modificación debido a que la alta capacidad de cómputo de este filtro no lo hace eficiente y asequible para colocarlo en el HDMI Out, por lo tanto, se optó por su implementación antes del “AXI Direct Memory Access”, elemento capaz de formar una sinergia adecuada con la IP desarrollada y aprovechar este diseño hardware correctamente.

7. **Implementación y pruebas en Jupyter**: Se han desarrollado y ejecutado pruebas experimentales utilizando notebooks de Jupyter para evaluar las implementaciones hardware y software de los filtros, se han introducido en código Python instrucciones que miden el tiempo de ejecución. Concretamente, se han usado la librerías "time" (para medir el tiempo transcurrido para calcular los frames por segundo y recursos) y "psutil" (para medir el uso de recursos de CPU y memoria), y las funciones contenidas en estas, medición de recursos (CPU y memoria): La función "measure_resources()" utiliza la biblioteca "psutil" para obtener el uso actual de CPU (`psutil.cpu_percent()`, que retorna el porcentaje de uso de CPU), y de memoria (`psutil.virtual_memory().percent`, que retorna el porcentaje de uso de memoria). Los frames por segundo se calculan con un simple cálculo haciendo la media del tiempo que se tarda en procesar cada frame, para ello, se utiliza la instrucción `time.time()`.

8. **Evaluación de tiempos de procesamiento y uso de recursos**: Medición y comparación de los tiempos de procesamiento entre las implementaciones en software y hardware. Y posterior análisis del uso de recursos y su impacto en la eficiencia y escalabilidad del sistema. El análisis de rendimiento entre las implementaciones de los filtros de vídeo (negativo, sepia, y Sobel) en software y hardware revela diferencias significativas. En términos de FPS, el filtro negativo se desempeñó mejor en software con una media de 12.0366 FPS frente a 11.4886 FPS en hardware. Sin embargo, los filtros sepia y Sobel mostraron una mayor eficiencia en hardware, con 11.7008 y 12.1887 FPS respectivamente, comparados con 9.2196 y 9.6284 FPS en software. La eficiencia del uso de CPU fue superior en hardware para todos los filtros, con el filtro sepia mostrando la mayor mejora (23.6393% en hardware frente a 57.5012% en software). En cuanto a la memoria, las diferencias fueron menores; el hardware fue más eficiente para los filtros negativo y Sobel, mientras que el filtro sepia tuvo un rendimiento ligeramente mejor en software. Estos resultados destacan la ventaja del hardware en aplicaciones de procesamiento intensivo de vídeo, mejorando la experiencia del usuario con un procesamiento más rápido y eficiente en la mayoría de casos, por lo que antes de realizar un diseño es conveniente hacer un estudio de las herramientas a utilizar y las posibles ventajas que pueden tener las distintas metodologías de diseño.

9. **Comparación de eficiencia entre software y hardware**: La evaluación y documentación de las ventajas en términos de eficiencia energética y capacidad de procesamiento de las implementaciones en hardware frente a las de software ha dado como resultado que no todas las implementaciones hardware ofrecen ventajas. En los filtros de baja complejidad, la implementación hardware no ofrece mejoras significativas. En algún caso incluso pueden verse reducidas las prestaciones. En los filtros con una mayor complejidad en su procesado (como es el caso del filtro Sobel) sí que se aprecian mejoras significativas en las implementaciones hardware.

Las contribuciones principales de este trabajo incluyen la creación de IPs personalizadas para filtros de vídeo, la integración efectiva de estas IPs en el diseño del sistema Pynq-Z2 y el desarrollo de un entorno de pruebas en Jupyter que permite comparar directamente el rendimiento del procesamiento de vídeo en hardware y software. Este enfoque ha permitido no solo validar la viabilidad técnica sino también cuantificar las ventajas e inconvenientes de la implementación en hardware.

Entre los retos técnicos más significativos, se encontraron la curva de aprendizaje asociada a las herramientas de desarrollo de hardware y las limitaciones inherentes al SoC Pynq-Z2. Estos desafíos subrayan la importancia de una planificación detallada y la necesidad de recursos educativos accesibles para desarrolladores.

Aunque los resultados son prometedores, la investigación se limitó a un conjunto específico de filtros y a una única plataforma hardware. Futuros estudios podrían ampliar el alcance a más algoritmos y comparar diversas plataformas SoC para obtener una comprensión más completa de la aplicabilidad de estas tecnologías.

7.1 Propuestas para futuros trabajos

Futuras investigaciones podrían explorar la optimización de los filtros para mejorar aún más el rendimiento y la eficiencia. Además, la implementación de filtros más complejos o avanzados tecnológicamente podría abrir nuevas vías de aplicación.

Dado el rápido avance en la tecnología SoC, investigar otras plataformas podría revelar oportunidades para mejorar el rendimiento o reducir los costos. Este enfoque también permitiría evaluar la portabilidad y adaptabilidad de las soluciones desarrolladas.

Personalmente, considero apasionante la tendencia actual que está adoptando el mundo de la tecnología para el desarrollo de hardware enfocado en su posterior aplicación software, enormemente relevante para casos como el desarrollo de GPUs por ejemplo. Este diseño combinando software y hardware es la solución que están adoptando las grandes empresas tecnológicas como NVIDIA o AMD, ya que acelerar los dispositivos con estos métodos es la opción más óptima para superar la barrera de las restricciones físicas que presentan los métodos y diseños realizados hasta la fecha.

8 REFERENCIAS

[1] – Microcontroladores en comparación con microprocesadores. Enlace consultado el 2 de febrero de 2024: https://learn.microsoft.com/es-es/azure/iot-develop/concepts-iot-device-types

[2] – Curso de S. Harish, “SOC architecture and design”. Consultado el 10 de octubre de 2023: https://es.slideshare.net/slideshow/soc-architecture-and-design/48351394

[3] – Imagen propia creada con Canva. Consultado el 12 de octubre de 2023: https://www.canva.com

[4] – Página oficial de AMD Xilinx, sección del producto Vitis HLS. Enlace consultado el 10 de octubre de 2023: https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html

[5] – Página oficial de AMD Xilinx, sección del producto Vitis. Enlace consultado el 10 de octubre de 2023: https://www.xilinx.com/products/design-tools/vitis.html

[6] – Página Open Source de GitHub de AMD Xilinx, sección Vitis Hardware Acceleration. Enlace consultado el 11 de octubre de 2023: https://xilinx.github.io/Vitis-Tutorials/2021-1/build/html/docs/Hardware_Acceleration/Feature_Tutorials/03-dataflow_debug_and_optimization/README.html

[7] – Blog Open Source sobre diseño electrónico de aceleradores hardware. Enlace consultado el 2 de febrero de 2024: http://systemonfpga.blogspot.com/2017/02/aceleracion-de-diseno-electronico-con.html

[8] – Página oficial de AMD Xilinx, sección del producto Vivado. Enlace consultado el 10 de octubre de 2023: Enlace consultado el: https://www.xilinx.com/products/design-tools/vivado.html

[9] – Página oficial de AMD Xilinx, sección para desarrolladores en Vivado. Enlace consultado el 10 de octubre de 2023: https://www.xilinx.com/developer/products/vivado.html

[10] – Página oficial de AMD Xilinx, sección para descargar Vivado ML Standard & Enterprise Edition. Enlace consultado el 10 de octubre de 2023: https://www.xilinx.com/products/design-tools/vivado/vivado-ml-buy.html

[11] – Página oficial de Jupyter, sección sobre los orígenes y la gobernanza del Proyecto Jupyter. Enlace consultado el 11 de octubre de 2023: https://jupyter.org/about

[12] – Página oficial de Jupyter. Enlace consultado el 11 de octubre de 2023: https://jupyter.org/

[13] – Página oficial de Jupyter, sección de documentación para usuarios. Enlace consultado el 11 de octubre de 2023: https://jupyter-notebook.readthedocs.io/en/stable/notebook.html

[14] – Página oficial de AMD Xilinx, portal de documentación. Enlace consultado el 4 de febrero del 2024: https://docs.xilinx.com/r/en-US/ug1579-microblaze-embedded-design/Introduction

[15] – Página oficial Open Source de PYNQ, sección Overlays. Enlace consultado el 4 de febrero de 2024: https://pynq.readthedocs.io/en/v2.1/pynq_overlays.html

[16] – Página oficial de AMD Xilinx, sección de dispositivos recomendados para universidades. Enlace consultado el 10 de octubre de 2023: https://www.amd.com/es/corporate/university-program/aup-boards/pynq-z2.html

[17] – Página inventario de dispositivos electrónicos Mouser Electronics. Enlace consultado el 10 de octubre de 2023 https://www.mouser.es/new/dfrobot/dfrobot-pynqz2-dev-board/

[18] – Zynq-7000 SoC Data Sheet: Overview. Enlace consultado el 11 de octubre de 2023: https://www.xilinx.com

[19] – PYNQ-Z2: Python Productivity for Zynq. Enlace consultado el 11 de octubre de 2023: https://www.pynq.io

[20] – Página oficial de PYNQ, "Getting Started with PYNQ". Enlace consultado el 4 de febrero de 2024: https://pynq.readthedocs.io

[21] – Página oficial de TUL, sección para comprar la placa, "PYNQ-Z2 Board User Guide. Enlace consultado el 10 de octubre de 2023: http://www.tul.com.tw/ProductsPYNQ-Z2.html

[22] – Página oficial de PYNQ, sección PYNQ Documentation: Proporciona información detallada sobre cómo utilizar Jupyter Notebooks para interactuar con el hardware de la placa Pynq-Z2. Enlace consultado el 14 de octubre de 2023: https://pynq.readthedocs.io/en/latest/

[23] – Página oficial de AMD Xilinx, sección Xilinx Documentation, Vitis High-Level Synthesis User Guide: Ofrece guías y tutoriales sobre el uso de Vitis HLS y Vivado para el diseño de lógica personalizada en la FPGA. Enlace consultado el 14 de octubre de 2023: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls

[24] – Proyecto perteneciente al curso de la Plataforma Udemy, "Learn Python Development with PYNQ FPGA: covers from Image Processing to Acceleration of Face Recognition Projects", impartido por Digitronix Nepal. Curso realizado el 20 de Noviembre de 2023: https://www.udemy.com/course/pynq-fpga-development-with-python-programming/

[25] – Repositorio oficial Open Source de Xilinx para las placas de desarrollo PYNQ. Enlace consultado el 14 de octubre de 2023: https://github.com/Xilinx/PYNQ

[26] – Proyecto Open Source sobre aceleradores hardware para filtros de procesado de vídeo. Enlace consultado el 15 de diciembre de 2023: https://github.com/aclich/PYNQ-Z2_sobel_filter_HDMI/tree/main