UNIVERSIDAD CENTRAL DE VENEZUELA
FACULTAD DE CIENCIAS - POSTGRADO EN CIENCIAS
Programa de Doctorado en Ciencias de la Computación

CODISEÑO Y PROGRAMACIÓN DE APLICACIONES PARALELAS EMBEBIDAS EN SISTEMAS DE COMPUTACIÓN HETEROGÉNEA RECONFIGURABLE: UN ENFOQUE BASADO EN ESQUELETOS ALGORÍTMICOS

Tesis Doctoral presentada ante la ilustre Universidad Central de Venezuela por el Magíster en Ciencias de la Computación y Doctor en Informática Carlos Alfonso Acosta León, para optar al título de Doctor en Ciencias, mención Ciencias de la Computación.

Tutora:
Dra. Rina Suros (UCV)

Jurado Principal:
Dr. Gerard Páez-Monzón (ULA)
Dr. Andrés Sanoja (UCV)
Dra. Ana Morales (UCV)
Dra. Dinarle Ortega (UC)

Caracas, 25 de Febrero de 2024
Caracas, 10 de Junio de 2024

Acta de Declaración de Autoría

Yo, Carlos Alfonso Acosta-León, egresado de la Escuela de Computación de la Facultad de Ciencias de la Universidad Central de Venezuela, de nacionalidad venezolana, titular de cédula de Identidad No. V-7.267.764, con la tesis doctoral titulada "CODISEÑO Y PROGRAMACIÓN DE APLICACIONES PARALELAS EMBEBIDAS EN SISTEMAS DE COMPUTACIÓN HETEROGÉNEA RECONFIGURABLE: UN ENFOQUE BASADO EN ESQUELETOS ALGORÍTMICOS",

Declaro bajo juramento que:

Primero: El trabajo de investigación es de mi exclusiva autoría.

Segundo: He cumplido y respetado las normas internacionales de citas y referencias para las fuentes consultadas. Por tanto, el trabajo de investigación no ha sido plagiado total ni parcialmente.

Tercero: El trabajo de investigación no ha sido autoplagiado, es decir, no ha sido publicada ni presentada anteriormente para obtener algún grado académico previo o título profesional.

Cuarto: Los datos presentados en los resultados son reales, no han sido falseados, ni duplicados, ni copiados y por tanto los resultados que se presentan en el trabajo constituyen un aporte real de la investigación.

Quinto: De identificarse fraude (datos falsos), plagio (información sin citar autores, auto plagio (presentar como nuevo algún trabajo de investigación propio que ya ha sido publicado), piratería (uso ilegal de información ajena) o falsificación (representar falsamente las ideas de otros) asumo las consecuencias y sanciones que de mi acción se deriven, sometiéndome a la normatividad vigente de la Universidad Central de Venezuela.

Carlos Alfonso Acosta-León
Cédula de Identidad: V-7.267.764
(Firma autógrafa)

UNIVERSIDAD CENTRAL DE VENEZUELA
FACULTAD DE CIENCIAS
COMISIÓN DE ESTUDIOS DE POSTGRADO

Comisión de Estudios de Postgrado

VEREDICTO

Quienes suscriben, miembros del jurado designado por el Consejo de la Facultad de Ciencias y el Consejo de Estudios de Postgrado de la Universidad Central de Venezuela, para examinar la **Tesis Doctoral** presentada por: **CARLOS ALFONSO ACOSTA LEÓN**, con cédula de identidad **Nro. V-7.267.764,** bajo el título **"CODISEÑO Y PROGRAMACIÓN DE APLICACIONES PARALELAS EMBEBIDAS EN SISTEMAS DE COMPUTACIÓN HETEROGÉNEA RECONFIGURABLE: UN ENFOQUE BASADO EN ESQUELETOS ALGORÍTMICOS"**, a fin de cumplir con el requisito legal para optar al grado académico de **DOCTOR EN CIENCIAS, mención CIENCIAS DE LA COMPUTACIÓN**, dejan constancia de lo siguiente:

1.- Leído, como fue, dicho trabajo por cada uno de los miembros del jurado, se fijó el día **lunes diez (10) de junio de 2024 a las 9:00 a.m.**, para que el autor lo defendiera en forma pública, lo que éste hizo en el Centro de Computación de la Facultad de Ciencias a través de la plataforma para realizar reuniones virtuales en línea Google Meet por Internet, mediante un resumen oral de su contenido, luego de lo cual respondió satisfactoriamente a las preguntas que le fueron formuladas por el jurado, todo ello conforme con lo dispuesto en el Reglamento de Estudios de Postgrado de la Universidad Central de Venezuela.

2.- Finalizada la defensa del trabajo, el jurado decidió, por considerar, sin hacerse solidario con las ideas expuestas por el Autor, que **se ajusta a lo dispuesto y exigido en el Reglamento de Estudios de Postgrado**.

Para dar este veredicto, el jurado estimó por unanimidad **APROBARLO**, por cuanto el trabajo examinado constituye un aporte original e importante en el *área de la Computación de Alto Rendimiento en Sistemas Heterogéneos Reconfigurables basados en FPGA, particularmente en el uso de técnicas de alto nivel para el codiseño y programación de aplicaciones paralelas embebidas en estos sistemas.*

Gerard A. Páez M. Dinarle Ortega

POSTGRADO EN CIENCIAS
DE LA COMPUTACION
Facultad de Ciencias
Universidad Central de Venezuela

En fe de lo cual se levanta la presente ACTA, a los diez (10) días del mes de junio del año 2024, conforme a lo dispuesto en el Reglamento de Estudios de Postgrado, actuó como Coordinadora del jurado la tutora, Dra. Rina Surós.

Aprobado en nombre de la Universidad Central de Venezuela por el siguiente jurado examinador:

Dr. Gerard Albert Páez Monzón
C.I. Nro. V-4.489.521
Universidad de Los Andes
(Jurado designado por el Consejo de la Facultad)

Dr. Andrés Fernando Sanoja Vargas
C.I. Nro. V-11.229.248
Universidad Central de Venezuela
(Jurado designado por el Consejo de la Facultad)

Dra. Ana Verónica Morales Bezeira
C.I. Nro. V-12.305.110
Universidad Central de Venezuela
(Jurado designado por el Consejo de Estudios de Postgrado)

Dra. Dinarle Milagro Ortega
C.I. Nro. V-8.611.660
Universidad de Carabobo
(Jurado designado por el Consejo de Estudios de Postgrado)

Dra. Rina Surós
C.I. Nro. V-3.811.397
Universidad Central de Venezuela
(Tutora)

POSTGRADO EN CIENCIAS DE LA COMPUTACION
Facultad de Ciencias
Universidad Central de Venezuela

NOTA MARGINAL (CERTIFICACIÓN)

Yo, Luis Manuel Hernández Ramos, con cédula de identidad Nro. V-6.766.769, **Coordinador del Postgrado en Ciencias de la Computación de la Facultad de Ciencias de la Universidad Central de Venezuela**, certifico que la Presentación y Defensa del Trabajo Doctoral se realizó de forma presencial, y a distancia por medios telemáticos síncronos, con la participación presencial de los profesores Dr. Andrés Fernando Sanoja Vargas, Dra. Ana Verónica Morales Bezeira y Dra. Dinarle Milagros Ortega; y con la participación en línea de los profesores Dr. Gerard Albert Páez Monzón y la Dra. Rina Surós (tutora), quienes autorizaron la inclusión de sus firmas digitales en la presente ACTA.

Firma: _____ **(Firma manuscrita)**

Gerard A. Páez M. Dinarle Ortega

POSTGRADO EN CIENCIAS
DE LA COMPUTACION
Facultad de Ciencias
Universidad Central de Venezuela

Dedicatoria

Dedico mi tesis doctoral principalmente a Dios, por ser la fuerza que me impulsa cada dia a proponerme y a culminar mis metas.

Por supuesto, también la dedico a las personas más importantes en mi vida con todo mi amor, cariño y pasión, quienes siempre me han apoyado y acompañado en mis logros.

Primeramente, la dedico a mi amada esposa Yemina, el ángel de mi corazón que ha cuidado de mí desde que la conocí. Gracias a su amor y apoyo constante e incondicional ha sido mi fortaleza en tiempos buenos y malos, siempre dándome ánimo, amor, cariño y consejo. Una de las dos mujeres que más amo y quiero en este mundo. Este trabajo es también su logro.

A mi madre, la Señora Haideé, mi primer ángel, que motivó mis anhelos, metas y esperanzas desde que era niño... y aún disfruta mis logros. La primera mujer que más amo y quiero en este mundo.

A mi hermanito menor Arturo, un hombre honesto, un luchador, excelente hermano y padre ejemplar, siempre dispuesto a ayudar a todos, hasta a aquellos que no lo merecen, con un corazón inmenso que Dios y la vida deben recompensar. Siempre he estado y estaré orgulloso de ti.

A mis hijas Mayeli y Aranza, y a mi nieta Mia; tres diamantes que adornan mi vida.

A mis hermanas Soyonara e Hiroshima, dos pedazos más de amor que complementan mi corazón.

Por último, y no menos importantes, a mis hermanas Maria Rosa, Gilda y Jenny; quienes ocupan un espacio en mi mente y en mi corazón desde antes de conocerlas.

Agradecimientos

Mi más sincero agradecimiento a mi tutora y amiga, Dra. Rina Surós, por su apoyo y confianza, sus consejos y correcciones me ayudaron a lograr esta etapa de mi trabajo de investigación. Gracias por su guía, le estaré agradecido por siempre.

Asímismo, aprovecho la oportunidad para agredecer al Dr. Gerard Páez-Monzón, profesor de la Universidad de Los Andes, por dedicar su tiempo a revisar mi trabajo, por sus comentarios y sugerencias de mejora, en fin, por sus invaluables recomendaciones.

Resumen

La presente tesis doctoral se enfoca en los Sistemas de Computación Heterogénea Reconfigurables (RHCS, Reconfigurable Heterogeneous Computing Systems). Estos sistemas se basan en dispositivos de computación de arreglos de compuertas programables por campo (FPGA, Field-Programmable Gate Arrays). En particular, el interés de este trabajo se centra en las técnicas de diseño y programación que facilitan el desarrollo de aplicaciones paralelas embebidas que explotan estos sistemas con alto rendimiento. Los sistemas de computación heterogénea reconfigurables (RHCS) se han utilizado para explotar paralelismo mediante el procesamiento acoplado y coordinado entre los FPGA y diferentes dispositivos de computación microprogramables como CPUs y GPUs. Sin embargo, estos sistemas tienen una alta complejidad de programación debido a los detalles asociados al paralelismo y al diseño digital de los FPGA. Esto hace que el proceso de implementar, al mismo nivel de abstracción, tareas y componentes en hardware y software de una aplicación embebida en los RHCS sea difícil de lograr. Es por esto, que el objetivo del presente trabajo doctoral es demostrar que mediante la integración de la técnica de codiseño de sistemas, el paradigma de algoritmo en hardware y el enfoque de esqueletos algorítmicos de Cole [Cole, 1989b] es posible proveer una herramienta con alto nivel de abstracción que facilita el codiseño y programación de aplicaciones paralelas embebidas en estos sistemas. Como solución se presenta una Interfaz de Programación de Aplicaciones Paralelas (Parallel Applications Programming Interface, PAPI), denominada SkeletonCoRe (Reconfigurable Skeletons Core for Parallel Programming). Esta PAPI provee al programador una librería o núcleo de esqueletos algorítmicos reconfigurables que expresan con alto nivel de abstraccion paralelismo implícito, estructurado y reconfigurable, encapsulando y ocultando los detalles de bajo nivel del paralelismo, así como el diseño y reconfiguración de los FPGA. Con esto se logra explotar de forma transparente y con alto rendimiento aplicaciones de procesamiento paralelo embebidas que aprovechen la arquitectura de Hardware y Software de los RHCS. Como pruebas de concepto se instancian dos esqueletos algorítmicos reconfigurables llamados *PipeSkeleton* y *TaskSkeleton* de SkeletonCoRe, que implementan los patrones de computación paralela "pipeline" y "master/slave", respectivamente. Estos esqueletos reconfigurables se proporcionan como plantillas de alto nivel en código OpenCL/C++ que heredan las propiedades de las funciones de orden superior, por lo cual toman parámetros de configuración para explotar paralelismo implícito. Estos permiten al programador escribir aplicaciones con alto rendimiento independientemente de la arquitectura del sistema heterogéneo disponible. Se ha definido la arquitectura jerárquica del PAPI con sus capas funcionales y las interacciones entre éstas. Luego, se ha especificado la estructura e interacción de los patrones de cómputo paralelo implementados como esqueletos y se han diseñado como funciones de orden supeior. Como demostración de SkeletonCoRe, se ha realizado un conjunto de pruebas usando ambos, *PipeSkeleton* y *TaskSkeleton*, con diferentes configuraciones sobre un sistema heterogéneo del tipo CPU, GPU y FPGA. Para la prueba de *PipeSkeleton* y *TaskSkeleton* se implementó una aplicación de procesamiento de imagen con 3 tareas coordinadas entre hardware y software. Para cada configuración se tomaron medidas del costo en tiempo de ejecución, entrada/salida y comunicación utilizados por el CPU, GPU y el FPGA. Los resultados demostraron que las configuraciones con núcleos implementados en FPGA y GPU, y las tareas de entrada/salida de datos implementadas por software en CPU se ejecutan más rápido y consumen menos recursos. Como conclusión, la PAPI SkeletonCoRe es una herramienta que proporciona al programador esqueletos reconfiguables con el suficiente nivel de abstracción para programar aplicaciones paralelas y mover fácilmente las funcionalidades entre las tareas de software y hardware durante la etapa de exploración de los espacios de diseño.

Palabras clave: Computación Heterogénea Reconfigurable, Paralelismo, Algoritmo en Hardware, FPGA, Esqueletos Algorítmicos, Co-diseño Hardware/Software.

Abstract

This PhD thesis focuses on Reconfigurable Heterogeneous Computing Systems (RHCS). These systems are based on Field-Programmable Gate Array (FPGA) computing devices. In particular, the focus of this work is on design and programming techniques that facilitate the development of parallel embedded applications that exploit these systems with high performance. Reconfigurable heterogeneous computing systems (RHCS) have been used to exploit parallelism through coupled and coordinated processing between FPGAs and different microprogrammable computing devices. However, these systems have a high programming complexity due to the details associated with parallelism and the digital design of FPGAs. This makes the process of implementing, at the same level of abstraction, hardware and software tasks and components of an embedded application in RHCSs difficult to achieve. Therefore, the aim of this doctoral work is to demonstrate that by integrating the systems co-design technique, the hardware algorithm paradigm and Cole's algorithmic skeleton approach [Cole, 1989b], it is possible to provide a tool with a high level of abstraction that facilitates the coding and programming of parallel applications embedded in these systems. The solution is presented as an Applications Programming Interface (PAPI), called SkeletonCoRe (Reconfigurable Skeletons Core for Parallel Programming). This PAPI provides the programmer with a library or reconfigurable algorithmic skeletons core. It expresses at a high level of abstraction implicit, structured and reconfigurable parallelism by encapsulating and hiding the low-level details of parallelism, as well as the design and reconfiguration of FPGAs. This enables the transparent and high performance exploitation of embedded parallel processing applications that take advantage of the RHCS hardware and software architecture. As a proof of concept, two reconfigurable algorithmic skeletons called *PipeSkeleton* and *TaskSkeleton* are instantiated, which are part of the SkeletonCoRe PAPI and implement the "pipeline" and "master/slave" parallel computing patterns, respectively. These reconfigurable skeletons are provided as high-level templates in OpenCL/C++ code, which take configuration parameters to exploit implicit parallelism. They allow the programmer to write applications with high performance regardless of the available heterogeneous system architecture. The PAPI hierarchical architecture with its functional layers and the interactions between them has been defined. Then, the structure and interaction of the parallel computing patterns that are implemented as skeletons have been specified and designged by using higher order functions. As a demonstration of the SkeletonCoRe PAPI, a set of tests have been performed using both, *PipeSkeleton* and *TaskSkeleton*, with different configurations on a heterogeneous CPU, GPU and FPGA system. For the *PipeSkeleton* and *TaskSkeleton* test, an image processing application was implemented with tasks coordinated between hardware and software. For each configuration, cost measurements were taken on execution time, input/output and communication used on the FPGA. The results showed that configurations with FPGA- and GPU-implemented cores and software-implemented input/output tasks on CPU run faster and consume fewer resources. As a conclusion, the SkeletonCoRe PAPI is a tool that provides the programmer with sufficient level of abstraction for applications parallel programming and easily move functionalities between software and hardware tasks during the exploration stage of application design spaces.

Keywords: Heterogeneous Reconfigurable Computing, Parallelism, FPGA, Algorithmic Skeletons, Hardware/Software Co-design.

Índice general

Índice de figuras

Índice de tablas

Índice de códigos fuente

Índice de algoritmos

Introducción

"*La religión es la cultura de la fe; la ciencia es la cultura de la duda.*"

Richard Feynman

"*La ciencia sin religión es coja, la religión sin la ciencia es ciega.*"

Albert Einstein

El presente trabajo de tesis doctoral se enfoca en los Sistemas de Computación Heterogénea Reconfigurables (RHCS, Reconfigurable Heterogeneous Computing Systems) basados en dispositivos de computación de arreglos de compuertas programables por campo (FPGA, Field-Programmable Gate Arrays). En particular, el interés se centra en las técnicas de diseño y programación que facilitan el desarrollo de aplicaciones embebidas en hardware y software que explotan estos sistemas con alto rendimiento.

Es conocido que desde hace años estamos en un contexto social y tecnológico cada vez más complejo y diverso donde áreas como la ingeniería, la investigación médica, la simulación científica, la exploración básica, la inteligencia artificial, los sistemas embebidos relanzados por la Internet de las Cosas, así como otras áreas que han sumado problemas tan exigentes que siguen demandando soluciones computacionales de alto rendimiento [Rodríguez et al., 2017]. Esto ha mantenido la tendencia a buscar un poder computacional no sólo mayor sino también barato, híbrido y eficiente.

En este sentido, la computacion de alto rendimiento (HPC, High-Performance Computing) [Robey and Zamora, 2021] es el área que utiliza técnicas, métodos y recursos computacionales, tanto en hardware como en software, para aumentar considerablemente la capacidad y rapidez de procesamiento. La HPC busca acelerar el tiempo de ejecución de las soluciones computacionales a problemas que exigen gran poder de cómputo, y que los sistemas de computación tradicionales no pueden lograr en tiempos útiles y razonables.

Un caso especial del área de la HPC son los sistemas de computación de arquitectura heterogénea (HACS, Heterogeneous Architecture Computing Systems). Estos sistemas, descritos por Zahran en [Zahran, 2019], integran distintos elementos de procesamiento, tanto estructural como funcionalmente, en un mismo sistema computacional de forma transparente al usuario. Esto permite incorporar capacidades de procesamiento especializado para acelerar tareas en un campo o área particular con el propósito de aumentar considerablemente el rendimiento.

En los HACS, la Computación Reconfigurable, concepto introducido por Gerald Estrin en 1959 [Bobda, 2007] y como bien se explica en [Babu and Parthasarathy, 2020], se refiere al procesamiento acoplado y coordinado entre dispositivos de hardware reconfigurable (FPGA) trabajando junto a microprocesadores de propósito general y microproceadores de propósito específico (CPU, GPGPU, DSP, etc.).

Es oportuno resaltar que dado que estos sistemas son de arquitectura híbrida es posible mezclar diferentes granularidades de procesamiento. Por ejemplo, los microprocesadores o CPU de múltiples núcleos (Multicore) son apropiados para el procesamiento de propósito general, de grano medio y grueso, basado en tareas y aplicaciones, respectivamente. Mientras que los microprocesadores gráficos o GPU de muchos núcleos (Manycore) son apropiados para el procesamiento de propósito específico de grano fino. Un caso especial son los FPGAs los cuales pueden ser usados para el procesamiento de propósito específico con alto rendimiento y diferentes granularidades, es decir, grano grueso (aplicaciones), medio (tareas) y fino (instrucciones).

La característica más importante de estos sistemas de cómputo es que combinan la flexibilidad del software de los microprocesadores con la flexibilidad del hardware, rapidez y alto rendimiento de los dispositivos reconfigurables [Koch et al., 2016]. Aquí, los Arreglos de Compuertas Programables por Campo (Field-Programmable Gate Arrays or FPGA) son los dispositivos de hardware reconfigurable más usados.

Estos sistemas son la base de la computación heterogénea reconfigurable de alto rendimiento (HPRC, High-Performance Reconfigurable Computing) y se presentan como una solución apropiada para lograr aplicaciones de alto rendimiento [Benkrid, 2014] por cuanto se aprovecha el paralelismo de diferente granularidad, tanto en hardware como en software, para acelerar tareas específicas de procesamiento con mayor rapidez y mayor eficiencia.

En este escenario, un sector de especial interés lo conforman las aplicaciones de alto rendimiento embebidas en sistemas, equipos y dispositivos que se utilizan en innumerables procesos y actividades de la industria, la ciencia y la tecnología [Khalgui and Hanisch, 2010]. Estas aplicaciones comprenden componentes, tareas o partes tanto en software como en hardware.

Estas aplicaciones, como se explica en [Barkalov et al., 2019], se caracterizan por realizar funciones dedicadas al control en tiempo real de sistemas, equipos o dispositivos específicos, estando integradas con componentes en software (microprogramas) y componentes en hardware (circuitos electrónicos) del equipo, funcionando cooperativamente. Además, y al contrario de lo que ocurre con las aplicaciones que se diseñan para sistemas de computación de propósito general que cubren un amplio rango de necesidades, los sistemas o aplicaciones embebidas se diseñan para cubrir necesidades específicas de procesamiento, y por ende se espera que tengan un mejor y alto rendimiento. Este punto constituye un foco importante de investigación donde se buscan métodos y herramientas para el diseño y programación de aplicaciones de alto rendimiento que exploten paralelismo en sistemas de computación heterogénea reconfigurables.

Como ya mencionamos, el problema de los sistemas de computación heterogénea reconfigurable es que tienen una alta complejidad de programación [Lai et al., 2021], lo cual es debido a que el desarrollo de aplicaciones que explotan paralelismo en estos sistemas necesitan la integración y coordinación de tareas corriendo en software y en hardware. Lo que implica que el programador deba prestar atención a detalles de bajo nivel asociados al paralelismo [Pacheco and Malensek, 2020], además de los detalles estructurales y funcionales asociados al diseño lógico de FPGA [Hajji et al., 2022].

Esta complejidad es la principal dificultad que encuentra el programador al emprender el desarrollo de aplicaciones paralelas embebidas en sistemas de computación heterogénea reconfigurable, y como consecuencia hay una limitada explotación de esta tecnología y del aprovechamiento de sus beneficios.

La reducción de esta complejidad de programación de los RHCS es necesaria para promover el uso de estos sistemas para desarrollar aplicaciones de alto rendimiento. Para ello, se deben proveer herramientas de diseño y programación con un mayor nivel de abstracción.

La presente tesis doctoral tiene como premisa de investigación la complejidad inherente asociada a la programación de los Sistemas de Arquitectura Heterogénea usados para Computación Reconfigurable basados en CPU, GPU y FPGA (Arreglos de Compuertas Programables por Campo) [Nedjah, 2016]. Esta complejidad, como ya se ha explicado, es objeto de preocupación en el ámbito científico y tecnológico por cuanto es aún un obstáculo que dificulta extender el uso de estos sistemas en la comunidad de programadores y como consecuencia ha limitado sus potenciales aplicaciones.

De ahí la motivación a buscar metodologías que reduzcan tal complejidad y simplifiquen el proceso completo de diseño, implementación y prueba de aplicaciones paralelas embebidas en plataformas heterogéneas reconfigurables [Pang and Membrey, 2017]. Además, La tesis doctoral se enmarca dentro de las lineas de investigación del Laboratorio de Computación Heterogénea de Alto Rendimiento del Centro de Computación Paralela y Distribuida (LabCHAR-CCPD) de la Escuela de Computación de la Universidad Central de Venezuela.

Particularmente, la investigación aborda uno de los objetivos del laboratorio concerniente al estudio y desarrollo de metodologías, tecnologías y herramientas con base en paradigmas que ayuden al programador durante el proceso de diseño e implementación de aplicaciones, sistemas y plataformas que exploten paralelismo. Esto puede aportar una importante contribución al área y lograr mayores avances y resultados en torno a este objetivo.

En consecuencia, en la presente investigación se propone como solución una herramienta que integra el enfoque de esqueletos algoritmicos de Cole, el paradigma de algorítmos en hardware y el método de codiseño de las secciones de hardware y software de sistemas. Es decir, la herramienta aprovecha la eficiencia de los algoritmos en hardware, así como su potencial adaptabilidad y rapidez de procesamiento usando el hardware de los FPGA, a lo cual se suma la capacidad de los esqueletos algorítmicos de encapsular paralelismo. Con todo esto se bueca proveer co-diseño y programación con el suficiente nivel de abstracción, a nivel de sistemas, para ocultar la complejidad de los HACS.

Se espera entonces, que lo anterior facilite al programador el uso de estas plataformas para explotar paralelismo implícito, estructurado y reconfigurable en el desarrollo de aplicaciones paralelas y así explotar los RHCS con alto rendimiento.

El documento se organiza de la siguiente manera: En el Capítulo 1, se presenta el problema, objetivos, justificación y motivación del trabajo doctoral. En el Capítulo 2, se revisa la literatura y se comentan algunos trabajos de investigación relacionados y relevantes que proponen métodos, técnicas y herramientas de diseño y programación usando el enfoque de Esqueletos Algorítmicos de Cole. También, se abordan los conceptos y defiinciones asociados a los paradigmas, métodos y técnicas que se integran en la funcionalidad de la herramienta propuesta. En el Capítulo 3, se presenta el diseño de la solución propuesta donde se explican la estructura y el funcionamiento de la Interfaz de Programación de Aplicaciones Paralelas *SkeletonCoRe* conformada, por ahora, de dos esqueletos reconfigurables. En los Capítulos 4 y 5, se muestra el diseño de estos dos esqueletos algorítmicos reconfigurables: PipeSkeleton y TaskSkeleton. En el Capítulo 6, se realizan pruebas de concepto de SkeletonCoRe usando los esqueletos reconfigurables PipeSkeleton y TaskSkeleton, donde además se hacen evaluaciones de funcionalidad y rendimiento. En el Capítulo 7 se analizan y discuten los resultados. Luego, se presentan algunas conclusiones y se recomienda trabajo futuro en el Capítulo 8. Al

final del documento se presentan las referencias bibliográficas empleadas y algunos anexos que pueden ser de interés.

Capítulo 1

Exposición del problema de investigación

"El científico no es aquella persona que da las respuestas correctas,
sino aquél quien hace las preguntas correctas."

Claude Lévi-Strauss

1.1 ANTECEDENTES DEL POBLEMA

Los avances tecnológicos en el campo de los sistemas y dispositivos de procesamiento han permitido acelerar considerablemente la ejecución de soluciones computacionales, y como consecuencia ha motivado también el estudio de nuevas técnicas, métodos y herramientas que faciliten el desarrollo de aplicaciones, sobre todo las que explotan paralelismo sobre estos sistemas. No obstante, el nivel de heterogeneidad de los sistemas de computación modernos aumenta gradualmente y se incrementa en el área de diseño de diferentes tipos de dispositivos o elementos de procesamiento, tal como se describen en [Makino, 2021].

El desarrollo de una aplicación, ya sea en hardware o en software [Koch et al., 2016, Snider, 2023], puede realizarse sobre diversas plataformas computacionales que pueden estar soportadas por sistemas microprogramables como los microprocesadores, por sistemas embebidos en forma de circuitos ASIC (Circuitos Integrados de Aplicación Específica, Application-Specific Integrated Circuit), por sistemas con hardware reconfigurables como los FPGA (Arreglos de Compuertas Programables por Campo, Field-Programmable Gate Arrays), etc.

En este contexto, el área de Sistemas de Computación con Arquitectura Heterogénea (HACS, Heterogeneous Architecture Computing Systems) propone una solución tecnológica que aborda esta diversidad, los cuales se describen suficientemente en [Hwu, 2016] y [Bell et al., 2018].

Esta área integra distintos elementos de procesamiento, tanto estructural como funcionalmente, en un mismo sistema computacional de forma transparente al usuario, incorporando capacidades de procesamiento especializado orientado a tareas en un campo o área particular con el propósito de aumentar considerablemente el rendimiento. Ejemplos de estos dispositivos o elementos de procesamiento se pueden

ver en la Figura 1.1, donde tenemos: microprocesadores, microcontroladores, GPU, ASIC, SOC, NOC, DSP, FPGA, etc.

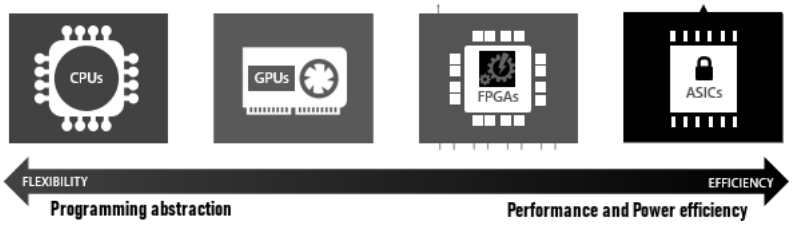


Figura 1.1: Visión comparativa de las caraterísticas de diferentes dispositivos de computación. Fuente: www.blockoperations.com.

El principio fundamental de los HACS es la cooperación funcional de cada tipo de dispositivo o elemento de procesamiento el cual aporta sus fortalezas para hacer posible su uso en estos sistemas de computación de forma efectiva y eficaz. Es decir, se aprovecha lo mejor de cada elemento para lograr una computación heterogénea que permita aumentar al máximo el rendimiento de las aplicaciones con un empleo eficiente de la energía.

Entre los HACS más populares están los sistemas de computación que incluyen hardware reconfigurable, y por ende explotan la Computación Reconfigurable, concepto introducido por Gerald Estrin en 1959 [Bobda, 2007]. Este concepto es la base de los Sistemas de Computación Heterogénea Reconfigurables (RHCS, Reconfigurable Heterogeneous Computing Systems).

De este modo, estos sistemas aprovechan el procesamiento acoplado, coordinado y cooperativo entre dispositivos con hardware reconfigurable (de propósito específico, como el ASIC y el FPGA) [Kirischian, 2016] [Hajji et al., 2022] y dispositivos con hardware microprogramable (de propósito general, como el CPU y el GPU) [Makino, 2021] (ver Figura 1.2). Además, estas plataformas permiten explotar potencialmente paralelismo en hardware y software y acelerar tareas específicas de procesamiento con alto rendimiento y eficiencia. Los FPGAs son el núcleo central de la Computación Reconfigurable, que es hoy día la alternativa más importante para desarrollar aplicaciones rápidas y de alto desempeño.

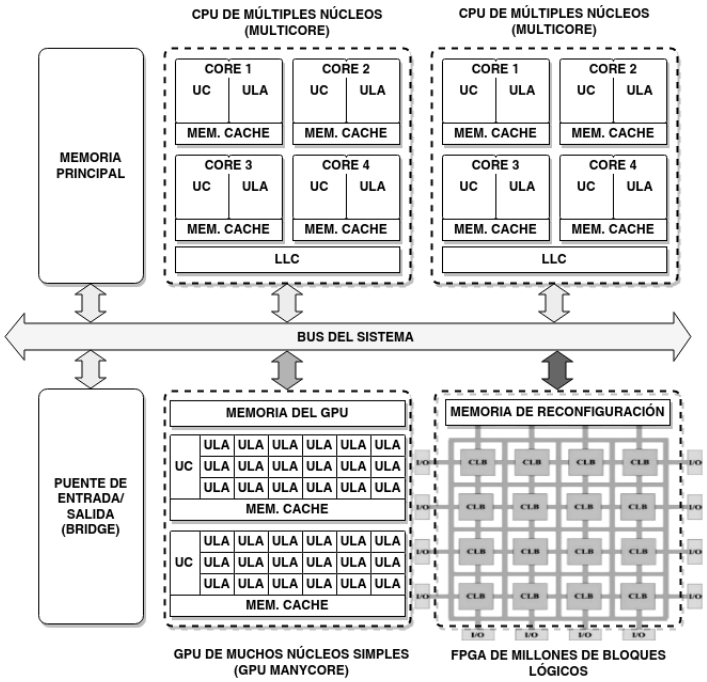


Figura 1.2: Arquitectura Heterogénea de un Sistema de Computación Reconfigurable CPU-GPU y FPGA. Fuente: Elaborado por el autor.

Es así que, por un lado el desarrollo de una aplicación que integra componentes funcionales (algoritmos o tareas) en hardware y en software [Koch et al., 2016, Snider, 2023] presupone algunas consideraciones al momento del desarrollo de una aplicación. Por ejemplo, en el caso de aplicaciones en software [Pacheco and Malensek, 2020], las operaciones e instrucciones asociadas al algoritmo o programa están generalmente codificadas usando un lenguaje de programación que al compilarlo produce un código de máquina que se ejecuta en un intérprete o autómata secuencial de hardware. Este autómata constituye el corazón de los sistemas de computación basados en un microprocesador de arquitectura fija (*paradigma de computación temporal o algoritmo en software*).

Por lo tanto, el microprocesador es en esencia una unidad microprogramable de hardware fijo que ejecuta diferentes programas secuencialmente en el computador, lo cual representa una solución más flexible dado que al cambiar las instrucciones del software, se modifica la funcionalidad del sistema sin la necesidad de cambiar el hardware (ver Figura 1.3a). Sin embargo, la desventaja de esta flexibilidad es que la rapidez de ejecución está por debajo de la rapidez de un ASIC [Makino, 2021]. Acá, la unidad central de procesamiento o CPU (Central Processing Unit) y la unidad de procesamiento gráfico o GPU (Graphic Processing Unit) siguen siendo los elementos de procesamiento microprogramable más comunes (ver Figura 1.3a).

Pero, por otro lado, el desarrollo de aplicaciones en hardware, como se muestra en [Wilson, 2016] y [Goossens, 2023], exige un lenguaje que permita la descripción de la funcionalidad asociada a un algoritmo en términos de componentes digitales o electrónicos de hardware (HDL, Hardware Description Language). Este lenguaje debe permitir la especificación de las instrucciones mediante operaciones directamente en circuitos lógicos basados en compuertas lógicas programables como el caso de los circuitos integrados de aplicación específica (ASICs) y los dispositivos de lógica reconfigurable.

Un dispositivo de lógica reconfigurable, núcleo básico de la unidad de procesamiento reconfigurable (RPU, Reconfigurable Processing Unit), se caracteriza por permitir la ejecución de varias funciones simultáneas,

en tiempo y espacio, implementadas directamente en hardware (*paradigma de computación espacial o algoritmo en hardware*), (ver Figura 1.3b). Aquí, los FPGA o arreglos de compuertas programables por campo (Field-Programmable Gate Arrays), como se explican en [Pang and Membrey, 2017] y [Goossens, 2023], son actualmente los dispositivos de hardware reconfigurable más usados.

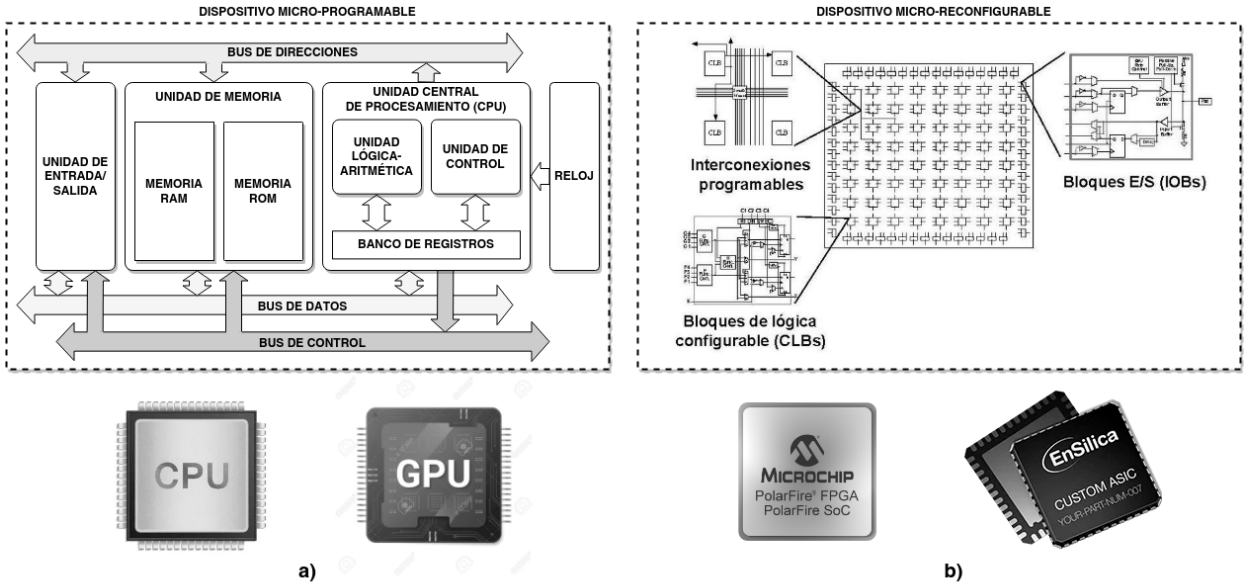


Figura 1.3: a) Unidad Microprogramable basada en un procesador de propósito general (CPU o GPU), b) Unidad Reconfigurable basada en un FPGA de propósito específico.
Fuente: Elaborado por el autor.

Los FPGA fueron inventados por Ross Freeman y Bernard Vonderschmitt en 1984 (co-fundadores de Xilinx) [de Schryver et al., 2013] buscando subsanar las limitaciones de los dispositivos lógicos programables o PLDs y como una evolución de los circuitos integrados de aplicación específica o ASIC.

En [Rodríguez et al., 2017] y en [Goossens, 2023] se resalta el poder que el hardware reconfigurable ha alcanzado debido a que integra las ventajas de los circuitos integrados de aplicación específica (ASIC, Application-Specific Integrated Circuit) y las ventajas de los procesadores de propósito general (CPU, Central Processing Unit). Así como los ASICs, los FPGA implican la implementación en hardware de funcionalidades típicas del software, y con ello paralelismo, alta capacidad de procesamiento y "reprogramabilidad/reconfigurabilidad", y en consecuencia, flexibilidad y rapidez para el diseño e implementación de aplicaciones.

En consecuencia, los FPGA llenan el espacio entre el hardware fijo de los ASIC y el software flexible que corre en la CPU dado que consigue un rendimiento potencialmente mayor que los microprocesadores, pero con un nivel más alto de flexibilidad que los ASIC. Estos han pasado de ser sencillos chips de lógica de acoplamiento a ser la base de la Computación Reconfigurable, brindando flexibilidad y reconfigurabilidad en la arquitectura heterogénea del sistema.

Si bien las primeras generaciones de FPGA fueron bastante limitadas en sus capacidades, en la actualidad se fabrican con las más modernas tecnologías de fabricación de circuitos integrados con una gran escala de integración (VLSI, Very Large Scale Integration). Esto ha favorecido el incremento progresivo de la complejidad de estos elementos de procesamiento y no sólo con millones de compuertas de lógica programable,

sino también con recursos específicos de hardware, con una amplísima gama de soluciones de conectividad tales como interfaces de entrada/salida, microcontroladores, módulos de memoria, lógica de interconexión e incluso microprocesadores RISC (Reduced-Instruction Set Computer), etc.

Actualmente se pueden construir con los FPGA aplicaciones completas de altísima complejidad haciendo de esta tecnología un elemento de computación transversal a múltiples disciplinas con soluciones a problemas en las más diversas ramas de la ciencia y la tecnología como la radioastronomía, emulación de hardware, bioinformática, criptografía, inteligencia artificial, etc [Thakare and Bhandari, 2023]. También, las universidades están extendiendo su uso dado que es una excelente herramienta didáctica para la enseñanza y para el diseño de prototipos [Barkalov et al., 2019].

La tecnología de FPGA continúa su impulso y apoyo por compañias importantes del área [Churiwala, 2017] como Xilinx, Atmel, Actel, Lattice Semiconductor, QuickLogic, Altera, Intel, AMD, etc. Se espera que el mercado de FPGA en todo el mundo aumente de 3.920 millones de dólares en el 2014 a 7.410 millones en el 2024[1]. Su uso se ha incrementando debido a la demanda emergente de sectores como el automotriz para proveer sistemas embebidos de asistencia al conductor (Driver Assistance Systems, ADAS); el de la electrónica de consumo para sus dispositivos móviles, al de Internet de las Cosas (Internet of things, IoT) para su uso en equipos electrónicos, al crecimiento de los centros de datos que demandan procesamiento masivo de datos (Big Data), Servicios de Computación en la Nube (Clod Computing). Todo esto ha conducido a una mayor presencia en el mercado mundial.

1.2 PLANTEAMIENTO DEL PROBLEMA

Durante los últimos años los procesadores de propósito general (CPU y GPU) han mejorado su rendimiento por la vía de la explotación de paralelismo de varios núcleos complejos y muchos núcleos simples [David A. Patterson, 2020] [Tanenbaum and Austin, 2012]. En esta dirección están los FPGA actuales cuya evolución estructural se ha adecuado para la explotación de procesamiento paralelo de datos y tareas. Además, los dispositivos FPGA modernos, en comparación con los GPUs (Graphic Processing Unit), proveen una razonable rapidez de procesamiento mientras consumen sólo una fracción de la energía de operación de los GPUs Many-Core y los CPUs Multi-Core [Rodríguez et al., 2017]. Estas capacidades de cómputo de los FPGAs han sido reconocidas por varias compañias tales como Intel y Microsoft, que han decidido usar FPGAs en vez de los GPUs como aceleradores en sus servidores de datos [Bobda et al., 2022].

Hay un interés actual en los FPGA, como se plantea en [Magyari and Chen, 2022], con respecto al potencial desarrollo de aplicaciones de alto rendimiento por la vía de la explotación del paralelismo inherente que comúnmente se encuentra en la mayoría de las soluciones a problemas complejos.

Los FPGA han despertado un interés en el área de la computación embebida de alto rendimiento [Barkalov et al., 2019,Thakare and Bhandari, 2023] por su capacidad de acelerar soluciones, en hardware y software, que explotan paralelismo, y por poder ser escalables a la complejidad de problemas masivos en datos e intensivos en cómputo [Wolf, 2014], buscando mejor equilibrio entre rendimiento, costo y eficiencia.

En este escenario, a la par del incremento de la complejidad y potencia de los recursos físicos disponibles en los FPGAs, ha estado también la evolución de las herramientas de síntesis, de simulación y de diseño a un nivel de abstracción de lógica digital, de lógica algorítmica y de lógica de sistemas que participan en las diferentes etapas del proceso de diseño.

[1] fuente: https://www.grandviewresearch.com/industry-analysis/fpga-market.

Por todo lo anterior, los FPGA siguen siendo hoy en día una opción viable, factible y muy interesante para el desarrollo de aplicaciones embebidas de alto rendimiento.

Sin embargo, el problema es que los sistemas de computación reconfigurable basados en FPGA presentan particularidades que no poseen los típicos sistemas de computación. El nivel de heterogeneidad de un sistema de computación reconfigurable del tipo CPU-GPU y FPGA introduce un proceso de diseño y desarrollo que no es uniforme y sus prácticas de programación son aún de alta complejidad [Koch et al., 2016].

1.3 FORMULACIÓN DEL PROBLEMA ESPECÍFICO

A pesar de las ventajas ofrecidas por los sistemas de computación reconfigurable y su rápido crecimiento, su uso todavía está restringido a un segmento bastante reducido de programadores de hardware [Koch et al., 2016]. Lo mismo sucede con la gran comunidad de programadores de software, quienes han permanecido alejados de esta tecnología.

La programación de aplicaciones orientadas a sistemas de computación reconfigurable basados en FPGA presenta sus retos, por cuanto el diseño e implementación de aplicaciones que explotan paralelismo e integran procesamiento tanto en hardware como en software agrega un nivel de complejidad adicional e importante. Esta complejidad se refiere a que el programador, al momento de desarrollar una aplicación paralela, debe prestar atención a los detalles de bajo nivel asociados al paralelismo [Pacheco and Malensek, 2020] tales como la creación, comunicación y sincronización de procesos, distribución de carga de trabajo, dependencia de recursos, entre otros.

Adicionalmente, si este programador quiere explotar paralelismo con alto rendimiento usando FPGA [Simpson, 2015] debe concentrarse también en detalles estructurales y funcionales asociados al diseño lógico como: compuertas lógicas, multiplexores, multiplicadores, registros, relojes, retardos de señal, conexiones, sincronización, eficiencia en el consumo de energía, etc. [Snider, 2023]. Ambos aspectos de diseño estan a diferentes niveles de abstracción, lo que dificulta el desarrollo de una aplicación paralela.

1.3.1 *PREGUNTAS DE INVESTIGACIÓN*

Partiendo del estado actual de las técnicas de desarrollo de software que están disponibles para programar de forma efectiva y eficiente aplicaciones paralelas embebidas con alto rendimiento, es posible plantearse las siguientes preguntas:

1. ¿Existen herramientas como lenguajes, técnicas o métodos que pueden utilizarse para proveer transparencia, abstracción y escalabilidad en el proceso de desarrollo de aplicaciones paralelas?

2. ¿Son estas herramientas también efectivas para ocultar los detalles del diseño digital en la implementación de aplicaciones paralelas en sistemas reconfigurables basados en FPGA?

3. ¿Es posible integrar estas herramientas en una solución que permita entonces el codiseño homogéneo de componentes de software y hardware de una aplicación paralela sobre sistemas heterogéneos de cómputo reconfigurable del tipo CPU-GPU-FPGA de forma integrada y con alto nivel de abtsracción?

Como se ha explicado, el programador aún se enfrenta en ésta área a un conjunto de complejidades que ameritan el empleo de diferentes estrategias y herramientas para su tratamiento, pero que todavía se usan de forma aislada.

Por ejemplo, el diseño e implementación de aplicaciones que integren partes tanto de hardware como de software agrega un nivel de complejidad adicional e importante a lo ya heredado por la programación paralela y su implementación en software (CPU-GPU) o en hardware (FPGA).

A pesar de todo esto, la Computación Heterogénea Reconfigurable no deja de ser un área computacional interesante para el desarrollo de aplicaciones complejas, de alto rendimiento, más rápidas y a un menor costo [Czarnul, 2018].

Esta área ya posee un conjunto de técnicas y herramientas de hardware que aunadas a las ya tradicionales técnicas en ingeniería del software podrían integrarse en nuevos métodos, metodologias y herramientas que reducirían aún más la complejidad de su programacioń y explotación.

Acerca de estas herramientas se hablará en el capítulo que aborda el marco conceptual preliminar que se usará como base teórica para sustentar la solución propuesta.

1.3.2 *FORMULACIÓN DE LA HIPÓTESIS DE INVESTIGACIÓN*

1.3.2.1 Problemática

Como se ha explicado anteriormente, la dificultad de desarrollar aplicaciones paralelas embebidas que integran componentes funcionales en hardware y software con el objetivo de explotar los sistemas heterogéneos de computación reconfigurable con alto rendimiento se debe principalmente a las complejidades asociadas al paralelismo y al diseño lógico de FPGA. Además, estas complejidades han obligado a aplicar diferentes estrategias y herramientas a diferentes niveles de abstracción, haciendo que el proceso de diseño y programación sea difícil, no uniforme, no homogéneo.

En este contexto, el enfoque de programación paralela basada en esqueletos algorítmicos ha sido exitoso cuando se ha aplicado al proceso de desarrollo de aplicaciones paralelas en software dado que ha permitido encapsular y ocultar sus complejidades, proveyendo en consecuencia paralelismo implícito y estructurado con suficiente abstracción al programador.

1.3.2.2 Hipótesis

Por lo tanto, en el presente trabajo se plenatea la siguinete hipóteis: “La Complejidad de programación de aplicaciones paralelas que exploten sistemas de computación heterogénea reconfigurable puede abordarse con éxito mediante una herramienta que integre el enfoque de esqueletos algorítmicos al proceso de codiseño y programación; y provea la suficiente abstracción que permita un desarrollo uniforme y homogéneo de las tareas de software y de hardware reconfigurable, lo que agrega al proceso de codiseño e implementación de dichas tareas el mismo nivel de abstracción, de forma escalable, reusable y eficiente”.

Esto presupone la factibilidad de construir una herramienta de programación que permita al programador manejar la complejidad inherente al codiseño e implementación de aplicaciones que explotan paralelismo estructurado y reconfigurable con la suficiente transparencia y abstracción que facilite la optimización y obtención de soluciones con alto rendimiento.

1.3.2.3 Propuesta de solución como Tesis

Con base en lo anterior, se propone como tesis doctoral el desarrollo de una herramienta de programación de aplicaciones paralelas en forma de biblioteca o librería que provea algoritmos paralelos en software y hardware, encapsulados como esqueletos algoritmicos que permitan al programador común un alto nivel de abstracción para explotar paralelismo de forma implícita y estructurada. Además, que facilite el proceso de

codiseño donde los componentes de hardware y software se diseñen e implementen de forma transparente y homogénea, facilitando al programador la migración de funcionalidades entre tareas de software y hardware. En fin, esta herramienta permitirá explotar los sistemas de computación heterogénea reconfigurable basados en miciroprocesadores microprogramables y FPGAs gracias a que esconde los detalles de bajo nivel asociados al paralelismo y la reconfiguración del FPGA. .

1.4 OBJETIVOS DE INVESTIGACIÓN

Con base en los argumentos explicados y la meta propuesta se presentan los siguientes objetivos de investigación:

1.4.1 *OBJETIVO GENERAL*

Desarrollar una Herramienta de Programación Paralela para Sistemas de Computación Heterogénea basados en dispositivos CPU, GPU y FPGA usando el enfoque de Esqueletos Algoritmicos que provea un alto nivel de abstracción para explotar paralelismo implícito, transparente y estructurado en el proceso de codiseño y programación de aplicaciones, y que permita el intercambio de funcionalidades entre tareas de software y hardware para explorar espacios de diseño y rendimiento.

1.4.2 *OBJETIVOS ESPECÍFICOS*

A partir del objetivo general se desprenden los siguientes objetivos específicos:

1. Diseñar algoritmos de procesamiento paralelo encauzado (pipeline) y maestro esclavo (master/slave) en forma de Esqueletos Algorítmicos Reconfigurables usando la noción de objetos para encapsular y ocultar los detalles asociados a la programación paralela y, el diseño y reconfiguración del hardware de los FPGA.

2. Implementar usando el API openCL en lenguaje C/C++ los Esqueletos Algorítmicos Reconfigurables como plantillas usando la noción de funciones de orden superior para pasar tareas como parámetros que determinen el comportamiento interno del esqueleto.

3. Evaluar la abstracción, funcionalidad y rendimiento de los Esqueletos Algorítmicos Reconfigurables implementados bajo diferentes configuraciones de diseño en un área o campo de aplicación específica.

1.5 IMPORTANCIA, RELEVANCIA Y VIGENCIA DE LA INVESTIGACIÓN

La Computación Reconfigurable es actualmente un paradigma de computación que ha abierto la puerta a nuevos paradigmas de computación en el ámbito académico, tecnológico y de investigación. Pero, su complejidad de programación, como ya se ha descrito, ha sido un obstáculo que ha impedido aprovechar sus beneficios.

Por esto, la importancia de reducir esta dificultad para promover su uso por el programador común, así como impulsar el aumento de la productividad para generar el desarrollo de más aplicaciones paralelas embebidas de alto rendimiento sobre estos sistemas.

1.5.1 *IMPORTANCIA Y RELEVANCIA DEL ÁREA DE INVESTIGACIÓN*

La reducción de la complejidad de programación de los RHCS es necesaria para continuar promoviendo el uso de estos sistemas, sus potenciales soluciones efectivas y eficientes, así como el desarrollo de herramientas

de diseño y programación con el nivel de abstracción adecuado. Esto contribuye a facilitar la programación de aplicaciones reconfigurables de alto rendimiento con la meta de mantener el interés en estos sistemas y sus potenciales aplicaciones.

El interés en los sistemas de computación heterogénea reconfigurable es cada vez mayor y se ha evidenciado cada año en Congresos, Conferencias y Simposios Internacionales especializados. Entre ellos se pueden citar, el *18th International Symposium International Symposium of Applied Reconfigurable Computing, ARC 2022* [Gan et al., 2022], el *32nd International Conference on Field-Programmable Logic and Applications, FPL 2022* [Zhang et al., 2022], el *30th IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM 2022* [Xu et al., 2022], entre otros. Estos reúnen la actividad de investigación y desarrollo tecnológico en el área, así como las diferentes aplicaciones de los FPGA y los sistemas de computación heterogénea.

1.5.2 *VIGENCIA DEL ÁREA DE INVESTIGACIÓN*

Se observa una activa investigación en centros y laboratorios de prestigiosas universidades del mundo liderados por reconocidos investigadores como Reiner Hartenstein (profesor del Departamento de Ciencias de la Computación de la Universidad Kaiserslautern de Tecnología, en Alemania), André DeHon (profesor del Departamento de Ingeniería Eléctrica y Sistemas de la Universidad de Pennsylvania, en Estados Unidos), entre otros; quienes han sido pioneros en la evolución, promoción, crítica y desarrollo de esta tecnología.

De igual manera, centros y laboratorios de investigación de compañias como Intel Co., AMD Inc., Xilinx, Altera, etc., han mantenido un interés sostenido en el uso, desarrollo y promoción de este tipo de tecnología basadas en los FPGAs. Asímismo, hay una alta producción de articulos de investigación sobre nuevas aplicaciones y un interés en liberar las patentes de hardwarte de esta tecnología.

1.6 ALCANCE, LIMITACIONES Y CONTRIBUCIÓN DE LA TESIS AL ÁREA DE ESTUDIO

1.6.1 *ALCANCE Y LIMITACIONES DE LA TESIS*

En el presente trabajo se tiene previsto sólo el desarrollo e implementación de una herramienta de programación de aplicaciones paralelas para sistemas heterogéneos reconfigurables en forma de una biblioteca, con al menos dos esqueletos algorítmicos implementados y usados como pruebas de concepto. Esta biblioteca provee una capa de abstracción y programación aprovechando la síntaxis, semántica y recursos de explotación de concurrencia proporcionados por el API OpenCL con un lenguaje base como C/C++, de uso común por la comunidad de programadores, para ocultar la expresión del paralelismo y la reconfiguración de los sistemas heterogéneos con aquitectura CPU-GPU y FPGA.

Adicionalmente, la experiencia y conocimiento del programador sobre el lenguaje de programación seleccionado ayuda a reducir la curva de aprendizaje de la herramienta. De esta manera el programador se concentra en los recursos de programación provistos en dicha capa de abstracción, lo que estimula una mayor eficiencia y productividad, debido al enfoque de programación de alto nivel.

Por lo tanto, queda fuera del alcance del presente trabajo doctoral el desarrollo de cualquier solución que implique la especificación, diseño o implementación de un lenguaje de programación de alto nivel para codiseño de aplicaciones paralelas embebidas para sistemas de computación heterogénea.

1.6.2 *CONTRIBUCIÓN DE LA TESIS AL ÁREA DE ESTUDIO*

La principal contribución de este trabajo de investigación es que la herramienta propuesta puede reducir la complejidad y distancia entre el programador común y el uso de los sistemas de arquitectura heterogénea

reconfigurables, lo cual se espera estimule un aumento en el desarrollo de aplicaciones embebidas y de alto rendimiento, así como la productividad de aplicaciones sobre estos sistemas.

También, se espera que la herramienta provea una base metodológica que guíe el proceso de codiseño de componentes de hardware y software de una aplicación paralela embebida, y ayude a ocultar y simplificar la implementación estructurada de paralelismo y reconfiguración con alto nivel, y facilite el movimiento de funcionalidades entre software y hardware durante la etapa de exploración de espacios de diseño de la aplicación.

Capítulo 2

Revisión de la Literatura y Trabajos Relacionados

"*Lo que se puede afirmar sin evidencia, puede ser descartado sin evidencia.*"

Christopher Hitchens

2.1 REVISIÓN DE TRABAJOS SOBRE ESQUELETOS EN HARDWARE O SOFTWARE

Hasta ahora son varios los proyectos que han propuesto herramientas como librerias, APIs, frameworks, ambientes y lenguajes de programación, etc., para el desarrollo de aplicaciones paralelas, tanto en el campo del diseño de aplicaciones para FPGA como para la programación de software paralelo.

Estos trabajos han involucrado diferentes tipos de soluciones para proveer distintos niveles de abstracción para el desarrollo de aplicaciones en software para microprocesadores o para el desarrollo de aplicaciones en hardware para FPGA. Pero, han sido soluciones orientadas a solo un dominio de implementación en software o en hardware de forma excluyente; y por lo tanto, no consideran el diseño coordinado (co-diseño) de aplicaciones construídas con componentes en hardware y en software, ni el intercambio de funcionalidades entre estos, que permita evaluar distintas configuraciones y seleccionar la solución de mayor rendimiento.

En este sentido, se han empleado métodos y técnicas de diseño de programas del portafolio de técnicas comunmente usadas en la Ingeniería del Software para el desarrollo de aplicaciones en software. Como ejemplo de estas técnicas tenemos el diseño y programación orientado a objetos, patrones de diseño, diseño basado en componentes, esqueletos algorítmicos, etc. Menos comunes son las técnicas para encapsular soluciones de hardware para FPGA, como el caso de la técnica de núcleos o bloques de propiedad intelectual (Intellectual Property Cores, IP Cores), etc. Todos ellos se han empleado para ocultar al programador los detalles y complejidades asociadas a la programación del dispositivo de procesamiento.

En el presente trabajo son de interés aquellos proyectos que se han concentrado en usar esqueletos algorítmicos como mecanismo de encapsulación del paralelismo tanto en software como en hardware.

Particularmente, en la literatura existen varios trabajos que proponen librerías, APIs o frameworks de patrones de cómputo paralelo encapsulados como esqueletos algorímicos.

Uno de estos primeros trabajos, pero usando esqueletos algorítmicos implementados en hardware, es de Benkrid and Crookes [Benkrid and Crookes, 2004]. Aquí, los autores diseñan una librería de esqueletos algorítmicos de operadores de imagen, para programación a bajo nivel, implementados en hardware de FPGA. Se usa la notación del lenguaje Prolog para describir las abstracciones de los esqueletos en el desarrollo de aplicaciones de procesamiento de imágenes. No obstante, tal programación no es fácil dado que Prolog usa la sintaxis y semántica de la programación lógica basada en reglas, poco familiar para el programador común. Otro trabajo similar al del grupo de Belfast que es necesario mencionar es el trabajo sobre "Skeletons and Asynchronous RPC for Embedded Data and Task Parallel Image" [Caarls et al., 2006], donde se usan Esqueletos Algorítmicos para la implementación de operadores de imagen, pero en software para ayudar en el desarrollo de aplicaciones que exploten paralelismo en el procesamiento digital de imágenes.

Un trabajo pionero y resaltante respecto al uso de esqueletos algorítmicos lo constituye la Librería de Esqueletos de Edimburgo "*The Edinburgh University Skeleton Library (eSkeL)*", del grupo de investigación del creador del paradigma "Skeletal Programming" o "Algorithmic Skeletons" [Cole, 1989a, Cole, 2004a], Dr. Murray Cole del Institute for Computing Systems Architecture (ICSA) de la Universidad de Edimburgo. En el proyecto eSkeL [Benoit et al., 2005] se propone una librería de esqueletos algoritmicos genéricos de software que explotan paralelismo de datos implementados en lenguaje C. Se proveen como una capa sobre la librería de pases de mensajes MPI (Message Passing Interface). De esta manera aprovechan su sintaxis y semántica para explotar paralelismo entre múltiples computadores en una red dedicada o Cluster.

Durante la estadía del autor del presente trabajo, realizando investigación en la Universidad de Edimburgo en 2008, tuve la oportunidad de trabajar en el área de esqueletos de hardware a nivel algorítmico [Acosta-León and Cole, 2008]. Un resultado interesante de dicha investigación fué el desarrollo de un patrón de cómputo paralelo pipeline (encauzamiento) encapsulado como un algoritmo en hardware de FPGA. Esta herramienta se orientó al diseño, simulación y verificación de aplicaciones de procesamiento de imágenes destinadas a reconfigurar un FPGA, pero pudiendo validar previamente su comportamiento mediante simulación sobre un computador. Para ello, se empleó el lenguaje de descripción de hardware JHDL (Java Hardware Description Language) para la especificación formal y verificación funcional del esqueleto.

Los trabajos mencionados anteriormente usan generalmente los esqueletos algoritmicos para desarrollar aplicaciones de software para CPU o de hardware para FPGA, pero no para sistemas de computación heterogénea que integran ambos dispositivos CPU y FPGA.

2.2 REVISIÓN DE TRABAJOS SOBRE ESQUELETOS HETEROGÉNEOS

Uno de los primeros trabajos orientado a sistemas de computación heterogénea proviene del grupo de investigación sobre paralelismo estructurado del Dr. Gorlatch, el proyecto SkelCL [Steuwer et al., 2011], que usa esqueletos algorítmicos para la programación de sistemas heterogeneos CPU-GPU. SkelCL es una librería de esqueletos implementados usando el API CUDA como dialecto de OpenCL, portables a diferentes GPU. Dicha libreria se compone de cuatro modelos de esqueletos paralelos: Map, Zip, Reduce y Scan, los cuales operan sobre vectores unidimensionales en CPU y GPU, aunque no de forma transparente. Sin embargo, no incorporan esqueletos de hardware, es decir, estan considerados solo para sistemas híbridos microprogramables del tipo CPU-GPU, sin considerar un dispositivo lógico reconfigurable como el FPGA.

Por otro lado, el proyecto HeteroCL de Lai et al. [Lai et al., 2019] es uno de los pocos trabajos donde se propone una infraestructura de programación compuesta por un lenguaje de dominio específico (DSL) basado en Python con compilación orientada a CPU, GPU y FPGA. Este proporciona abstracción mediante una programación que desvincula la especificación del algoritmo del hardware de la arquitectura de computación. También, permite al programador explorar el rendimiento de manera sistemática con implementaciones de hardware usando plantillas de matrices sistólicas y stenciles de flujo de datos. No obstante, la herramienta no considera el co-diseño homogéneo de las tareas o componentes en hardware y software de una aplicación paralela al mismo nivel de abstracción.

En el mismo sentido, en un trabajo más reciente en el área, Ernstsson et al. proponen SkePU [Ernstsson et al., 2021], un framework de programación de sistemas heterogéneos CPU-GPU de código abierto para CPUs multinúcleo y sistemas multi-GPU. Son plantillas en C++ con esqueletos de paralelismo de datos, con soporte para la ejecución en sistemas multi-GPU tanto con CUDA como con OpenCL y ejecución en clusters.

Estos proyectos antes mencionados tampoco incorporan esqueletos de hardware para arquitecturas heterogéneas del tipo CPU-GPU-FPGA.

Recientemente, en Octubre del año 2022, el autor del presente trabajo hizo una ponencia de un artículo de investigación [Acosta-León and Suros, 2022] en una importante Conferencia Latinoamericana sobre Computación de Alto Rendimiento. En este artículo se hace un primer avance de la tesis doctoral proponiendo un esqueleto algorítmico reconfigurable basado en el patrón de cómputo paralelo encauzado o “pipeline” que permite el codiseño y programación de aplicaciones paralelas compuestas de tareas en software y tareas en hardware sobre sistemas de computación heterogénea reconfigurable del tipo CPU-GPU y FPGA. Esta herramienta de programación paralela es una librería de Esqueletos Heterogéneos Reconfigurables, como parte del proyecto “SkeletonCoRe” del Laboratorio de Computación Heterogénea de Alto Rendimiento del Centro de Computación Paralela y Distribuida (LabCHAR-CCPD) de la Escuela de Computación de la Universidad Central de Venezuela. Este proyecto aborda el problema del desarrollo de aplicaciones paralelas a alto nivel, de forma integrada y homogénea desde las perspectivas de hardware y software. En este sentido, se ha aprovechado el paralelismo implícito y estructurado, los algoritmos embebidos en hardware y software, y la reconfigurabilidad de los FPGAs aplicando el Paradigma de Esqueletos Algorítmicos al Codiseño Hardware/Software.

2.3 CONSIDERACIONES FINALES DEL CAPÍTULO

Con base en los trabajos revisados se observa que existen herramientas que proveen un nivel de abstracción superior a los lenguajes de descripción de hardware (HDL). Sin embargo, son pocas las soluciones orientadas a facilitar el co-diseño y programación de aplicaciones paralelas que integran partes que interactúan en hardware y software en plataformas de computación heterogéneas reconfigurables basadas en FPGA. Además, algunos trabajos revisados suponen que el programador tiene algún conocimiento de la arquitectura del sistema de computación reconfigurable, de los detalles asociados al paralelismo y del diseño de FPGA.

Capítulo 3

Marco Conceptual Preliminar

"*Lo que se puede afirmar sin evidencia, puede ser descartado sin evidencia.*"

Christopher Hitchens

3.1 INTRODUCCIÓN

Además de la revisión de trabajos relacionados al área de investigación, en el presente capítulo se hace un resumen de algunos elementos teóricos y tecnológicos que son útiles para entender la propuesta de solución de la presente tesis doctoral. Aunque existen varias técnicas de abstracción de alto nivel, en este trabajo de investigacion son de interés aquellas que permiten ocultar y encapsular el paralelismo. A partir de estas técnicas de diseño y programación se han elegido los esqueletos algorítmicos de Cole, la técnica de co-diseño hardware/software y el paradigma de algoritmo en hardware usando FPGA. Estos constituyen el fundamento para el diseño de la herramienta propuesta que las integra en una solución o API denominado SkeletonCoRe. A continuación se hace la suficiente explicación del background teórico que fundamenta la herramienta propuesta.

3.2 CODISEÑO DE APLICACIONES HARDWARE/SOFTWARE

La tecnología VLSI hace posible construir poderosos componentes programables de propósito general (microprocesadores) y también componentes reconfigurables de propósito general o especial (FPGAs or ASICs - Application-Specific Integrated Circuits).

Un Sistema de Computación Reconfigurable consiste de una mezcla de dispositivos VLSI de procesamiento en hardware y software. Desafortunadamente, las aplicaciones destinadas a estos sistemas poseen tareas o componentes que generalmente se diseñan separadamente, usando diferentes herramientas y técnicas de diseño, y por personas diferentes.

Por ello, son de interés aquellas técnicas donde el diseño de las secciones de hardware y software de una aplicación se haga de forma integrada. La idea principal es que la especificación del sistema se haga de

tal manera que diferentes partes de la aplicación puedan implementarse indistintamente en hardware o en software.

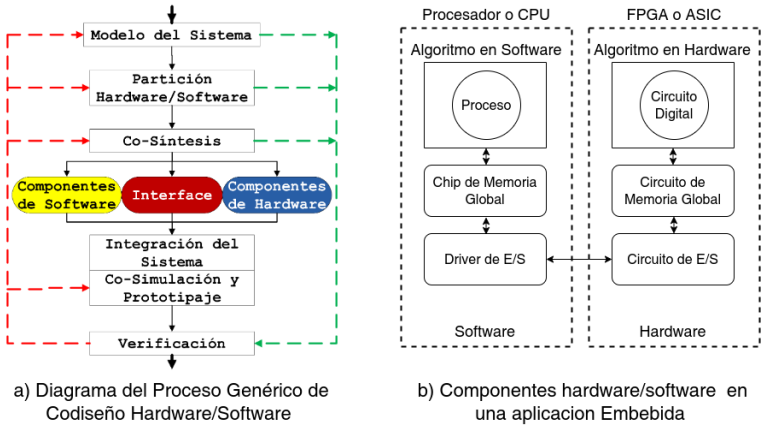
3.2.1 *DEFINICIÓN DE CODISEÑO HARDWARE/SOFTWARE*

En torno al área de desarrollo de aplicaciones, tanto en software como en hardware, ya existen diversas técnicas y metodologías en Ingeniería de Software. Una de estas técnicas de abstracción útil se le denomina Codiseño de Hardware/Software [Cardoso and (auth.), 2011, Schaumont, 2013]. Esta técnica integra el desarrollo cooperativo de aplicaciones conformadas por componentes que interactúan en hardware y software, aunque no garantiza que estos componentes se desarrollen al mismo nivel de abstracción. Esto dificulta la migración de funcionalidades entre software y hardware (ver Figura 3.1). Además, estas partes, tareas o componentes deben ser capaces de interactuar usando interfaces de comunicación, independientemente de su implementación.

El codiseño hardware-software [Murti, 2022] tiene como propósito mejorar el proceso de desarrollo de un sistema embebido que cuenta con partes de hardware, orientadaos por ejemplo a dispositivos reconfigurables como FPGAs (Field Programmable Gate Array) y partes programables por software, orientadas a microprocesadores y DSPs. Esta tendencia, plantea un cambio de paradigma con respecto al modelo tradicional, a pesar de que su uso aún no está extendido en la industria. Esto hace que cada vez más el mercado tecnológico demande que los fabricantes doten a sus herramientas de capacidades de diseño de FPGA (Arreglo de Compuertas Programables por Campo) con capacidades para el desarrollo de software, es decir, con partición automática del sistema en componentes de software y hardware, que posibiliten la transformación de una idea inicial en un diseño final, de forma rápida, eficiente, fiable y, por supuesto, con bajo costo y consumo.

3.2.2 *METODOLOGÍA DE CODISEÑO HARDWARE/SOFTWARE*

En la Figura 3.1a se puede observar la metodología genérica que se ha venido utilizando tradicionalmente para el diseño de aplicaciones embebidas. El método [Murti, 2022] parte de una especificación informal del sistema, luego se deciden las partes del sistema que se implementarán en hardware o software, así como sus interfaces de comunicación (Figura 3.1b). Este proceso de partición o división de los componentes o partes en hardware y en software es realizado manualmente por los diseñadores en base a su experiencia previa. Posteriormente, se desarrollan por separado cada componente de hardware y software utilizando metodologías de diseño de hardware (FPGA o ASIC) y software (CPU-GPU), respectivamente. Continúa con la integración de las secciones de hardware y software desarrollados y su posterior co-simulacion y verificación funcional.



a) Diagrama del Proceso Genérico de Codiseño Hardware/Software

b) Componentes hardware/software en una aplicacion Embebida

Figura 3.1: a) Metodología para el Codiseño Hardware/Software. b) Componentes hardware/software en una aplicacion embebida. Fuente: Elaborado por el autor

La tecnología de sistemas digitales es cada vez más compleja y la metodología tradicional de diseño de estos sistemas no es adecuada para sistemas complejos, ya que el aumento de la complejidad genera los siguientes problemas: **a)** El espacio de diseño crece de forma considerable. Por tanto, es muy poco probable que el particionado hardware/software realizado manualmente por un diseñador sea óptimo; y **b)** Se incrementa la aparición de errores al realizarse la integración final de hardware y software debido a que el proceso de diseño se realiza a partir de una especificación informal del sistema.

En los últimos años se han propuesto diferentes metodologías de codiseño hardware/software, que aportan algunas soluciones para resolver los problemas anteriores: **a)** La exploración del espacio de diseño y la optimización del particionado hardware/software pueden ser mejorados mediante estrategias que provean abstraccion en cuanto al encapsulamiento del comportamiento de los componentes de hardware y software del sistema; y **b)** Los errores detectados en la fase de integración del hardware y software pueden disminuirse diseñando los sistemas a partir de lenguajes formales de especificación que permiten la simulación y verificación de los diseños.

En definitiva, las herramientas para la especificación y desarrollo cooperativo deben considerar como recursos la abstracción y el encapsulamiento de los componentes en hardware y software producto de su partición en el codiseño de sistemas.

3.2.3 *APLICACIONES DEL CODISEÑO*

Un uso actual y más frecuente de la técnica de codiseño se debe al advenimiento de la Internet de las Cosas, paradigma que ha estimulado el desarrollo de sistemas embebidos [Murti, 2022], también conocidos como sistemas “empotrados”, “incrustados” o “integrados”. Estos son sistemas de computación diseñados para realizar funciones específicas, y cuyos componentes de software y hardware se encuentran integrados en una placa base con dispositivos CPUs, FPGAs y/o ASICs. Aquí, el procesamiento central del sistema se lleva a cabo a través de un microprocesador o microcontrolador, donde se ejecutan las tareas de software, apoyado por tareas realizadas por hardware especializado, que incluye además interfaces de comunicación y entrada/salida, así como una memoria de tamaño reducido en el mismo chip.

3.3 COMPUTACIÓN HETEROGÉNEA RECONFIGURABLE Y FPGAS

Los Sistemas de Computación con Arquitectura Heterogénea (HACS, Heterogeneous Architecture Computing Systems), los cuales se describen suficientemente en [Hwu, 2016], integran distintos elementos

de procesamiento, tanto estructural como funcionalmente, en un mismo sistema computacional. Esta arquitectura heterogénea se presenta de forma transparente al usuario o programador, incorporando capacidades de procesamiento especializado orientado a tareas en un campo o área particular con el propósito de aumentar considerablemente el rendimiento.

Un caso particular de estos sistemas HACS lo conforman los Sistemas de Computación Heterogénea Reconfigurable (HPRC, High-Performance Reconfigurable Computing). Estos combinan la flexibilidad del software sobre microprocesadores (CPUs y GPUs) con el alto rendimiento y flexibilidad del hardware de los dispositivos reconfigurables (FPGAs). En estos sistemas los Arreglos de Compuertas Programables por Campo (Field-Programmable Gate Arrays o FPGAs) son los dispositivos de hardware reconfigurable más comunmente usados.

3.3.1 *DEFINICIÓN, ESTRUCTURA Y FUNCIONAMIENTO DE UN FPGA*

3.3.1.1 Definición de FPGA

Un Arreglo de Compuertas Programables por Campo o FPGA [Kirischian, 2016, Hajji et al., 2022] es un dispositivo semiconductor construido como una plantilla VLSI (Very Large Scale Integrated) reconfigurable y reusable, estructurada como una malla regular de bloques o elementos de computación (computation blocks o **CB**) embebidos en una red de bloques o elementos de entrada/salida (I/O blocks o **IOB**) y de interconexión (interconnection blocks o **INB** y bus lines), cuyas funcionalidades son programables vía software (ver Figura 3.2). Están constituidos por bloques de lógica cuya interconexión y funcionalidad se puede programar. La lógica programable puede reproducir desde funciones tan sencillas como las llevadas a cabo por una compuerta lógica o un sistema que combine hasta complejos sistemas en un chip.

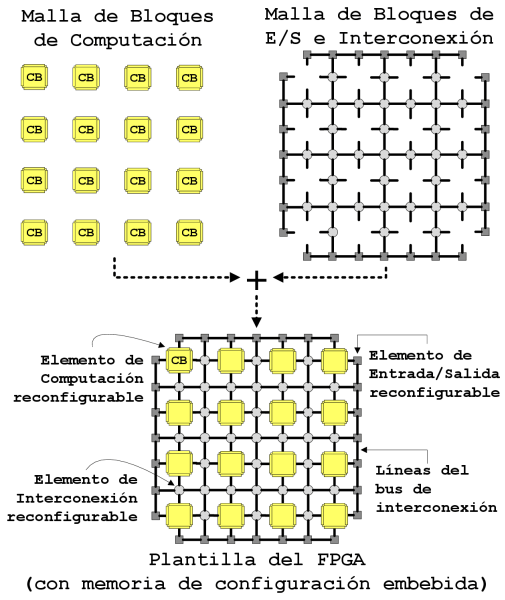


Figura 3.2: Estructura genérica de un FPGA: Matriz de Procesamiento y Matriz de interconexión y Entrada/Salida para conformar una Plantilla de diseño de hardware. Fuente: Elaborado por el autor.

Una tendencia reciente ha sido combinar los bloques lógicos e interconexiones de las FPGA con microprocesadores y periféricos relacionados como interfaces de red y entrada/salida, etc., para formar un sistema programable en un chip. Además, muchas FPGA modernas soportan la reconfiguración parcial

del sistema, permitiendo que una parte del diseño sea reprogramada, mientras las demás partes siguen funcionando. Este es el principio de la computación reconfigurable, o de los sistemas reconfigurables.

Actualmente, la complejidad y densidad de los FPGA es de cientos, miles y hasta de millones de bloques lógicos que permiten implementar microprocesadores, microcontroladores, procesadores DSP, filtros de imagen, interfaces de red y entrada/salida de alta velocidad, y mucha memoria integrada en chip, etc., (ver Figura 3.4). Están a disposición del programador para desarrollar aplicaciones o algoritmos más complejos en hardware. Las aplicaciones donde más comúnmente se utilizan los FPGA incluyen sobre todo en aquellas aplicaciones que requieren un alto grado de paralelismo.

3.3.1.2 Estructura de un FPGA

Los FPGAs están compuestos de un conjunto de módulos lógicos relativamente independientes entre sí, que pueden interconectarse para formar un circuito mayor. Estos módulos lógicos son bloques básicos de computación (CB, Computing Block) que pueden ser grandes bloques configurables o pequeños elementos de función fija formados por algunas compuertas como se observa en las Figuras 3.3 a) y b). Para implementar una función lógica o algoritmo en el FPGA estos CB se interconectan por medio de canales programables o configurables, mediante un proceso conocido como enrutamiento, como se ve en la Figura 3.3 c). El enrutamiento consiste básicamente en determinar, ya sea en forma manual o a través de herramientas de cómputo, una estrategia de interconexión eficiente.

Los FPGA se estructuran en varios tipos de Bloques Lógicos, y que son configurables cuando esta es programada, como se observa en la Figura 3.3:

- **Look Up Tables (LUT):** Estos bloques almacenan una lista, predefinida por el usuario, de salidas y entradas lógicas. Se carga con valores relevantes para la FPGA según sus instrucciones específicas. Una LUT es como una RAM que se carga cada vez que se conecta la FPGA. Cuando se configura una FPGA, los bits de la LUT se cargan con unos (1's) o ceros (0's), en función de la configuración de la Tabla de Verdad con los valores lógicos deseados. En lugar de conectar numerosas puertas lógicas, para crear la tabla de verdad, esto se simula en un tipo especial de RAM. Las LUT con cuatro a seis bits de entrada son ampliamente utilizadas.

- **Multiplexores (MUX):** Son circuitos combinacionales con varias entradas y una única salida de datos. Disponen de entradas de control capaces de seleccionan una sola de las entradas de datos existentes. La FPGA los emplea como dispositivo para recibir varias entradas y transmitirlas por un medio de transmisión compartido, dividiendo el medio de transmisión en múltiples canales.

- **Flip-Flops tipo D (DFF):** Son registros binarios que se utilizan para sincronizar la lógica y guardar estados lógicos entre ciclos de reloj dentro de una FPGA. En cada flanco de reloj, un flip-flop bloquea el valor 1 o 0 en su entrada y mantiene ese valor constante hasta el siguiente flanco de reloj. Se utiliza para mantener el estado dentro del chip.

- **Full Adders (FA):** Realiza operaciones aritméticas en la FPGA, sumando o restando dígitos binarios. Para transformar toda la multitud de Bloques Lógicos en la configuración correcta y ejecutar la aplicación, el diseño comienza definiendo las tareas requeridas en la herramienta de desarrollo y luego compilándolas en un archivo de configuración que contiene información sobre cómo conectar los CLB y otros módulos. El proceso es similar a un ciclo de desarrollo de software, excepto que el objetivo es diseñar el propio hardware en lugar de un conjunto de instrucciones para ejecutar en una plataforma de hardware predefinida.

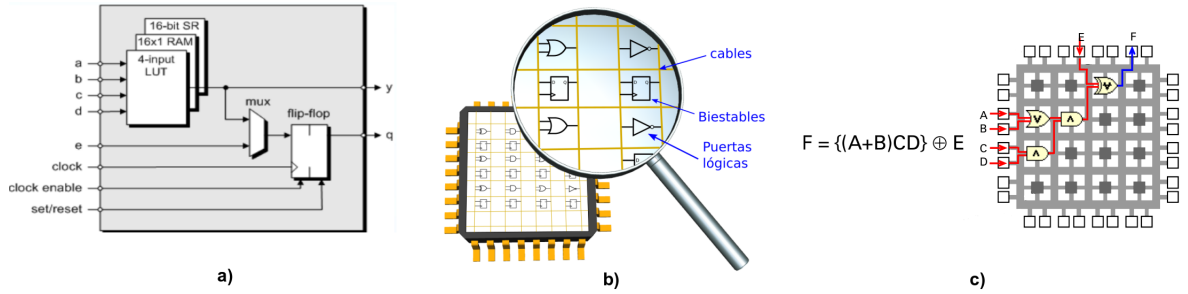


Figura 3.3: a) Celda básica de cómputo (CB), b) Estructura lógica-digital de un FPGA, c) Circuito lógico implementado en FPGA. Fuente: Elaborado por el autor.

3.3.1.3 Funcionamiento de un FPGA

La tecnología FPGA permite diferentes formas de funionamiento cambiando la configuración de los bloques y sus conexiones:

- **Reconfiguración total o estática:** donde la funcionalidad se fija en tiempo de compilación y no cambia durante el funcionamiento de éste. Cada vez que se realiza una nueva configuración, todo el FPGA se actualiza. Se tiene que detener la operación del FPGA, configurarlo todo y volver a ponerlo en ejecución. Es para dispositivos con acceso secuencial a la memoria de configuración. Hay penalizaciones de tiempo de configuración.

- **Reconfiguración parcial o dinámica:** durante la ejecución ciertas partes del FPGA se configuran con una nueva funcionalidad, sin detener la ejecución del resto del FPGA.

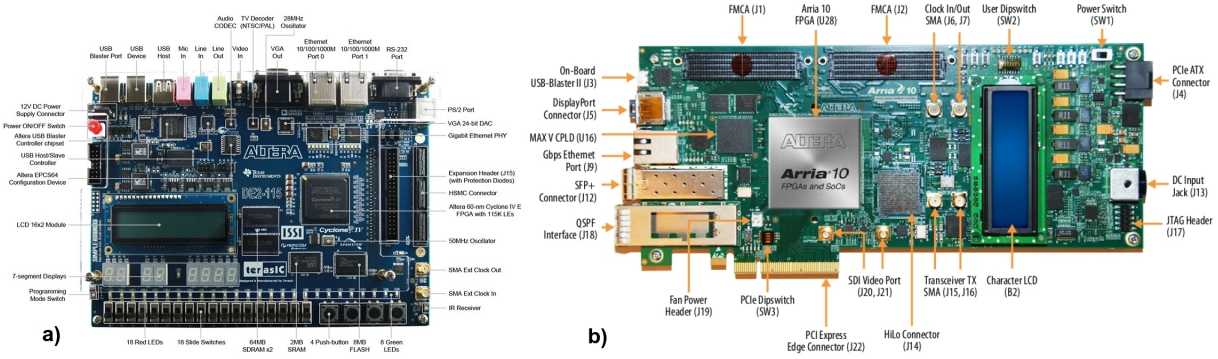


Figura 3.4: Tarjetas FPGA para prototipos: a) Tarjeta de desarrollo FPGA Altera DE2-115 (Conexión USB), b) Tarjeta Intel Arria 10 (Conexión PCie). Fuente: www.wikipedia.org

3.3.2 *APLICACIONES DE LOS FPGA*

Cualquier circuito de aplicación específica puede ser implementado en una FPGA, siempre y cuando ésta disponga de los recursos necesarios. Las aplicaciones donde más comúnmente se utilizan las FPGA incluyen a los DSP, radio definido por software, sistemas aeroespaciales y de defensa, prototipos de ASICs, sistemas de imágenes para medicina, sistemas de visión para computadoras, centros de datos, reconocimiento de voz, bioinformática, emulación de hardware de computador, entre otras. Cabe destacar que su uso en otras áreas es cada vez mayor, sobre todo en aquellas aplicaciones que requieren un alto grado de paralelismo como en

la inteligencia artificial. Los FPGA también se utilizan para prototipar nuevas CPU y GPUs, es decir, para probar nuevas funcionalidades a nivel de hardware antes de su implementación en un procesador.

El mercado actual de los FPGA se ha colocado en un estado en el que hay dos grandes productores de FPGA de propósito general (Xilinx e Intel) y un conjunto de otros competidores que ofrecen dispositivos FPGA con características específicas. Los principales fabricantes son: Xilinx Inc., Intel Co., Altera Co., Lattice Semiconductor Co., Actel (actualmente Microsemi), Atmel, Achronix Semiconductor Co., Quicklogic Corporation, entre otras.

3.3.3 *DISEÑO Y PROGRAMACIÓN DE UN FPGA*

La tarea del programador es definir la función lógica que realizará cada uno de los CLB del FPGA, seleccionar el modo de trabajo de cada IOB e interconectarlos o enrutarlos (ver Figura 3.2 f).

Para ello, el programador descarga los bits de configuración (todos estan contenidos en un archivo bitstream) para modificar la funcionalidad de los bloques lógicos y las interconexiones entre ellos. Generalmente los bits se almacenan en una memoria SRAM incluida en cada bloque lógico, que es quien controla la interconexión y funcionalidad de los elementos lógicos de cada bloque lógico.

El diseñador cuenta con la ayuda de entornos de desarrollo especializados en el diseño y prototipado de sistemas a implementarse en una FPGA. Un diseño se puede llevar a cabo, ya sea como un diagrama esquemático, o haciendo uso de un lenguaje de programación especial.

En un intento de reducir la complejidad y el tiempo de desarrollo en fases de prototipado rápido, y para validar un diseño en HDL, existen varias propuestas y niveles de abstracción del diseño. Los niveles de abstracción superior son los funcionales y los niveles de abstracción inferior son los de diseño al nivel de componentes hardware básicos.

Como una forma de manejar esta complejidad y facilitar el uso de los FPGA es que surgen los Lenguajes de Descripción de Hardware (HDL, Hardware Description Language) [Hauck and DeHon, 2007]. Estos lenguajes de programación especializados son usados para describir la estructura, diseño y operación de circuitos electrónicos y en especial de sistemas lógicos digitales.

Para ello, la síntesis (ver Figuras 3.5 y 3.6) de un sistema digital supone un proceso previo diseño y compilación del código HDL para generar un archivo denominado ”netlist” con las especificaciones de bajo nivel de la configuración de los componentes y estructura física (compuertas lógicas, flip-flops, multipĺexores, etc.). También, se genera el mapa de interconexiones, temporización y sincronización de reloj y las máscaras para configurar el circuito integrado del FPGA. Finalmente, se implementa el diseño convirtiendo el archivo ‘netlist’ en una serie de bits que configuran físicamente las interconexiones, estructura y funcionalidad lógica del FPGA, denominado ‘bitstream’.

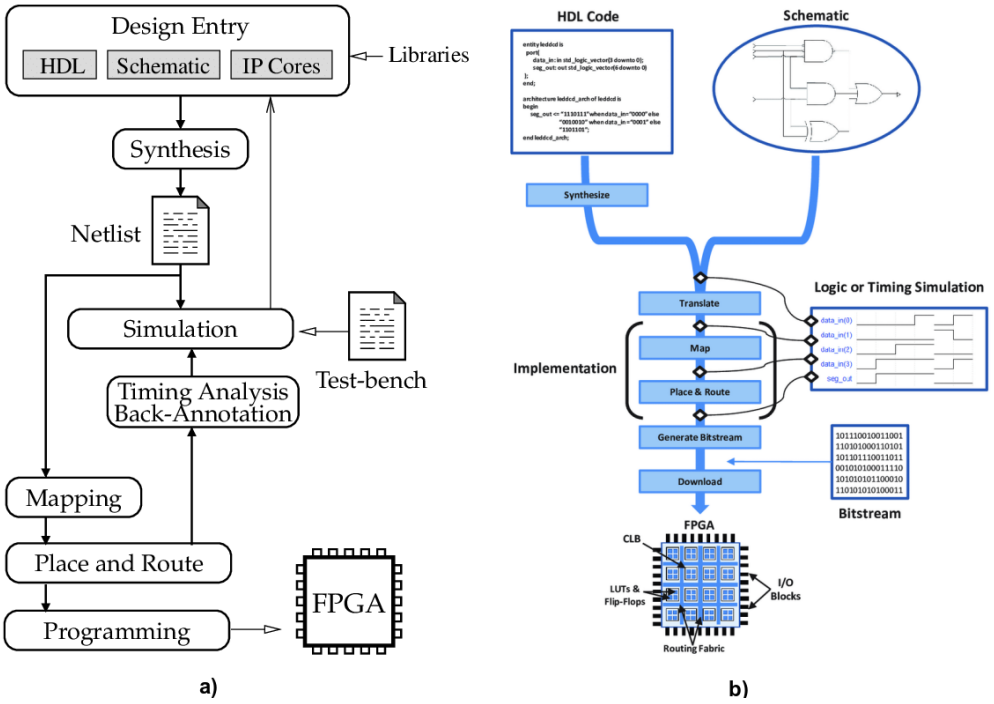


Figura 3.5: Dos formas de visualizar el proceso de diseño de un sistema digital en un FPGA. Fuente: www.wikipedia.org.

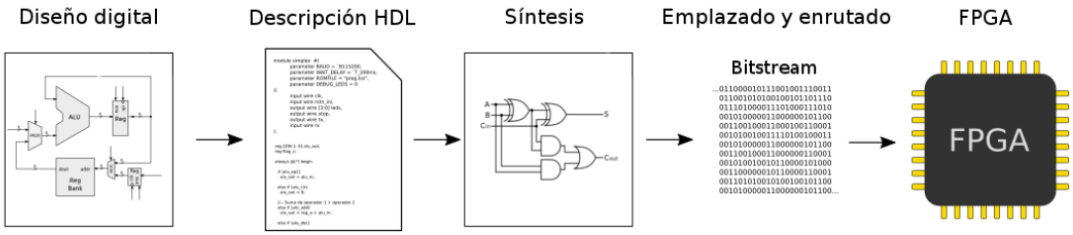


Figura 3.6: Proceso de abstracción de diseño e implementación de un sistema digital en un FPGA. Fuente: Elaborado por el autor.

Los HDLs son muy parecidos a los lenguajes de programación de alto nivel como C, Java o Python, pero se diferencian de estos en su sintáxis y semántica la cual incluye explícitamente la noción de tiempo y expresiones para explotar concurrencia a muy bajo nivel. Algunos lenguajes de descripción de hardware muy usados para la programación de FPGAs son: VHDL, Verilog, JHDL y SystemC.

Además, los principales fabricantes de FPGAs proveen herramientas para hacer más sencillo el proceso de diseño de una FPGA. Así, Xilinx Inc. ofrece la macroherramienta Xilinx Vivado o ISE Design Suite, que consiste en un conjunto de herramientas integradas destinadas al diseño y desarrollo FPGA, entre otros. Este entorno está destinado al diseño de un sistemas SoC o HDL respectivamente. Contiene las siguientes herramientas:

- EDK: herramienta para diseño de sistemas con microprocesador.
- SDK: herramienta para el diseño del software de un sistema.
- System Generator: basado en Simulink, se emplea para diseñar sistemas desde un nivel de abstracción superior al de las herramientas anteriores.

- ISim: es el simulador para la depuración del sistema diseñado.
- ChipScope: sirve para la monitorización de las señales internas de una FPGA a través del cable JTAG sin necesidad de emplear un osciloscopio.

Por otra parte, el principal software de Intel para el diseño de FPGAs es Quartus, una macroherramienta de la que se pueden destacar las siguientes herramientas:

- Quartus: para realizar sistemas usando HDL o usando un lenguaje de alto nivel como OpenCL.
- SOPC Builder: empleado para diseñar sistemas con microprocesador SoC.
- Qsys: empleado para conectar a un nivel alto bloques, sistemas, IPs, etc.
- DSP Builder: herramienta basada en Simulink para facilitar el diseño de sistemas.

Como una alternativa a los HDL han emergido lenguajes que no están orientados al diseño de hardware, pero permiten programar FPGAs a un nivel de abstracción lo suficientemente alto que permite al programador ignorar los detalles asociados a la implementación física de la lógica digital, dejando la optimización y construcción del hardware al compilador. Ejemplos de estos lenguajes son OpenCL, openMP/openACC, entre otros.

3.4 Nociones de Computación Paralela y Esqueletos Algorítmicos

En los últimos años han proliferado cada vez más problemas que requieren soluciones que demandan un alto poder de cómputo para su resolución. La existencia de estos problemas complejos computacionalmente exige la fabricación de sistemas de computación mas potentes para poder solucionarlos en un tiempo útil y adecuado.

Por eso, los dispositivos y sistemas de computación secuenciales [Hennessy and Patterson, 2017] han utilizado técnicas como la segmentación encauzada (pipeline processing), la computación vectorial (vector processing), computación maestro/esclavo (master/slave processing), etc., para incrementar su rendimiento y productividad. También se aprovechan de los avances tecnológicos, especialmente en la integración de circuitos, que les permite acelerar su rapidez de cómputo.

Tradicionalmente, un programa, software o aplicación [Pacheco and Malensek, 2020] consiste de una secuencia de operaciones e instrucciones asociadas a un algoritmo que describe la solución al problema. Estos generalmente están codificados usando un lenguaje de programación de bajo o alto nivel que al compilarlo produce un código de máquina que se ejecuta secuencialmente en un sistema de computación basado en un microprocesador microprogramable de arquitectura fija (*paradigma de computación temporal o algoritmo en software*).

Acá, la unidad central de procesamiento o CPU (Central Processing Unit) y la unidad de procesamiento gráfico o GPU (Graphic Processing Unit) **(ver Figura 1.3a)** son los elementos de procesamiento más comunes. Estas unidades microprogramables poseen un hardware fijo que ejecuta diferentes programas en el computador, lo cual representa un enfoque más flexible dado que al cambiar las instrucciones del software, se modifica la funcionalidad del sistema sin la necesidad de cambiar el hardware **(ver Figura 1.3a)**. Sin embargo, la desventaja de esta flexibilidad es que la rapidez de ejecución está limitada por el reloj de trabajo y la arquitectura fija del sistema [Makino, 2021].

Por otro lado, el paralelismo es una técnica de computación basada en el principio “Divide y vencerás” el cual consiste en dividir un problema en varios pequeños subproblemas, resolverlos al mismo tiempo y luego

integrar sus soluciones. Este enfoque implica la tarea de distribuir la carga de trabajo entre los elementos de procesamiento disponibles, lo cual resuelve el problema en menos tiempo.

Es por ello que el paralelismo [Hennessy and Patterson, 2017] a través de la multiplicidad de elementos de procesamiento más simples ha pasado a ser la técnica más barata y eficiente para obtener mayor rendimiento. Así, con varios dispositivos de cómputo y computadores menos avanzados de costo inferior puede obtenerse buen rendimiento al integrarlos en un sistema multiprocesador o multicomputador. Además, en un computador secuencial la rapidez de cómputo siempre estará limitada a la tecnología existente, por lo que tiene un tiempo limitado de vida realmente útil. Por otro lado, en un sistema multicomputador es fácil aumentar el rendimiento si se aumenta el número de procesadores que lo forman, lo cual mejora por encima de un computador secuencial. Los sistemas de multiples computadores como los clusters de estaciones de trabajo, los procesadores de múltiples núcleos complejos (multicore) y los procesadores de muchos núcleos simples (manycore) ofrecen una buena relación calidad/precio frente a los super computadores. Todo ello es posible debido al avanzado desarrollo tecnológico en la arquitectura de los microprocesadores, y en las redes de interconexión, que permiten buenos indices de rapidez de procesamiento asi como buenas velocidades de comunicación y ancho de banda, respectivamente.

Por todo esto surge la necesidad de desarrollar sistemas de cómputo y métodos de programación paralela para mejorar significativamente el rendimiento obtenido en máquinas secuenciales. No obstante, la programación paralela es mas difícil de implementar que la programación secuencial, pero aún así vale la pena dado que sin duda ofrece mejor rendimiento y eficiencia.

3.4.1 *TIEMPO, RENDIMIENTO, EFICIENCIA Y GANANCIA DE VELOCIDAD USANDO PARALELISMO*

Un parámetro que se suele dar para caracterizar los sistemas con varios elementos de procesamiento (multiprocesadores) es el rendimiento máximo del sistema [Grama et al., 2003]. Habitualmente este rendimiento se suele calcular como el número de procesadores del sistema multiplicado por el rendimiento de cada uno de los procesadores.

Cuando el sistema opera al máximo rendimiento todos los procesadores están realizando un trabajo útil; ningún procesador está detenido (ocioso) y ningún procesador ejecuta instrucciónes extras que no estén en el algoritmo original. En este estado de rendimiento máximo o pico todos los n procesadores están aportando al rendimiento efectivo del sistema y la velocidad de procesamiento viene incrementada por un factor n.

No obstante, el estado de rendimiento máximo o de pico es un estado raro que difícilmente se puede alcanzar debido a varios factores que introducen ineficiencia. Algunos de estos factores son los siguientes:

- Retrasos introducidos por las comunicaciones entre procesos.
- La sobrecarga de trabajo debida a la necesidad de sincronizar la carga de trabajo entre los distintos procesadores.
- La pérdida de eficiencia cuando algún procesador se queda sin trabajo para realizar y queda ocioso.
- La pérdida de eficiencia cuando uno o más procesadores realizan algún esfuerzo inútil.
- El costo de procesamiento para controlar el sistema y la programación de las operaciones.

Como se explica en [Schmidt et al., 2018], las medidas más utilizadas para determinar el rendimiento de un sistema ya sea paralelo o secuencial (en hardware o software) son: el tiempo de ejecución o rapidez de procesamiento de una aplicación, y la productividad (throughput) o número de tareas que es capaz de procesar y completar por unidad de tiempo. Además, se emplean otras métricas como: la aceleración

(speedUp), eficiencia del sistema, utilización, redundancia, consumo de energia, cantidad de operaciones en punto flotante por segundo, etc. A continuación se revisan algunos de estos parámetros usados para evaluar el rendimiento de un sistema computacional:

a) Tiempo de ejecución de un programa: El tiempo de ejecución es la medida de rendimiento más intuitiva. Es un parámetro que permite expresar la rapidez del algoritmo sin compararlo con otro.

En el caso de un programa secuencial, consiste en el tiempo transcurrido desde que se inicia su ejecución hasta que finaliza. En el caso de un programa paralelo, el tiempo de ejecución es el tiempo que transcurre desde el comienzo de la ejecución del programa en el sistema paralelo hasta que el último procesador culmine su ejecución [Grama et al., 2003].

En un algoritmo paralelo el tiempo de ejecución depende del tamaño del problema, la cantidad de procesadores y las comunicaciones. Para sistemas paralelos con memoria distribuida el tiempo paralelo con p procesadores, T_{par}, se determina de modo aproximado mediante la fórmula 3.1:

$$T_{par} = T_{cpu} + T_{com} + T_{solap} \tag{3.1}$$

donde: T_{cpu} es el tiempo de cálculo o cómputo aritmético en CPU, es decir, el tiempo que tarda el sistema multiprocesador en hacer las operaciones aritméticas; T_{com} es el tiempo de comunicación, o sea, el tiempo que tarda el sistema multiprocesador en ejecutar transferencias de datos; y T_{solap} es el tiempo de solapamiento, que es el tiempo que transcurre cuando las operaciones aritméticas y de comunicaciones se solapan o realizan simultáneamente. Este tiempo de solapamiento suele ser muchas veces imposible de calcular tanto teórica como experimentalmente, sin embargo influye bastante en el tiempo total de ejecución T_{par} del algoritmo paralelo. No obstante, la dificultad de su cálculo obliga a realizar la aproximación de la fórmula 3.2:

$$T_{par} = T_{cpu} + T_{com} \tag{3.2}$$

b) Aceleración (Speed-Up) y Eficiencia del Sistema: El Speed-Up o S_p para p procesadores, es el cociente entre el tiempo de ejecución de un programa secuencial T_{sec}, y el tiempo de ejecución de la versión paralela de dicho programa en p procesadores o T_{par}. Dado que pueden haber distintas versiones secuenciales, se elige el T_{sec} de la versión secuencial más rápida. Este índice indica la ganancia de velocidad que se ha obtenido con la ejecución en paralelo. El factor de mejora del rendimiento (speed-up o aceleración) se define en la ecuación 3.3:

$$S(n) = \frac{T_{sec}(1)}{T_{par}(n)} \tag{3.3}$$

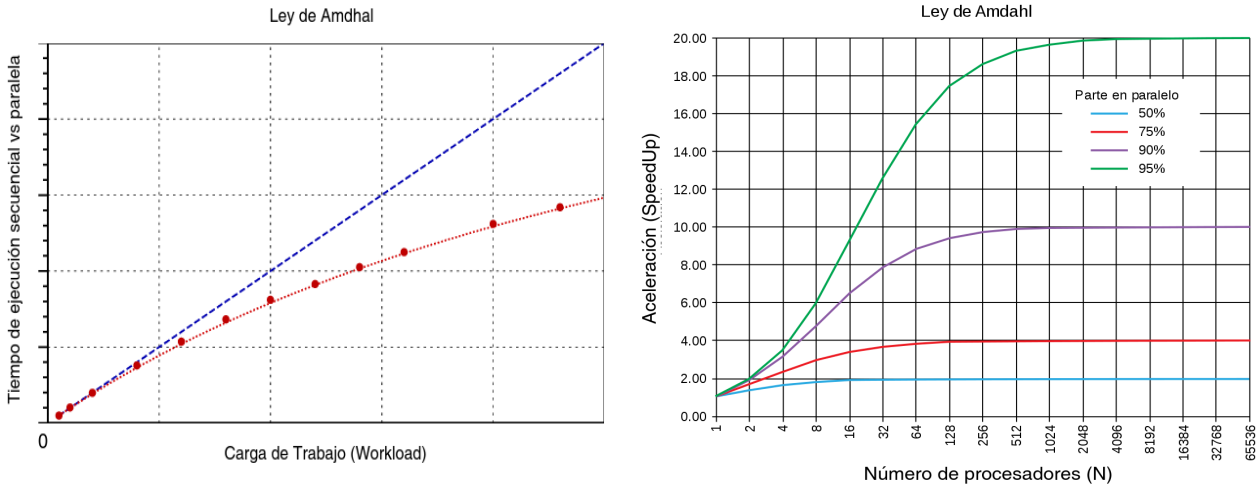
Es oportuno mencionar que es una buena práctica comparar el tiempo de ejecución paralela con respecto al tiempo de ejecución de la mejor solución o algoritmo secuencial. Sin embargo, en [Grama et al., 2003] se recomienda que dado que muchas veces es difícil conseguir o probar el mejor algorítmico secuencial para tal comparación, también es útil correr la solución paralela en un sólo procesdor o usando un sólo proceso o hilo para simular la versión secuencial del mismo.

Por otro lado, la eficiencia es el cociente entre la aceleración (Speed-Up) y el número de procesadores. Significa el grado de aprovechamiento de los procesadores para la resolución del problema. El valor máximo

que puede alcanzar es 1, que significa un 100 % de aprovechamiento. Así, la ecuación 3.4 muestra la eficiencia como una comparación del índice de ganacia de rapidez (speed-Up) obtenido con respecto al valor máximo. Dado que $1 \leq S(n) \leq n$, tenemos $1/n \leq E(n) \leq 1$. Por un lado, la eficiencia más baja $E(n) \to 0$ corresponde al caso en que el programa se ejecuta en un único elemento de procesamiento o procesador de forma secuencial. Mientras que la eficiencia máxima $E(n) = 1$, se obtiene cuando todos los elementos de procesamiento se utilizan para ejecutar las porciones paralelas o simultáneas del programa durante todo el periodo de ejecución.

$$E(n) = \frac{S(n)}{n} = \frac{T_{sec}(1)}{n * T_{par}(n)} \leq 1 \tag{3.4}$$

Por tanto, al explotar paralelismo se busca aumentar el rendimiento del procesamiento de la solución a un problema específico, es decir, obtener una mejora en la rapidez de ejecución de las tareas ejecutándolas en un sistema con varios elementos de computación con respecto a los sistemas secuenciales o monoprocesadores [Robey and Zamora, 2021]. El SpeedUp representa entonces la ganacia que se obtiene en la version paralela del programa respecto a la version secuencial del mismo (ver Figura 3.7).



(a) Tiempos de ejecución secuencial versus paralelo.

(b) Aceleración ($T_1/T_p(N)$

Figura 3.7: Gráficas con ejemplos de curvas de comportamiento ideal asociados al rendimento y la aceleración en paralelismo. Fuente: https://hpc.llnl.gov.

c) Escalabilidad: Un sistema es escalable para un determinado rango de procesadores $[1...n]$, si la eficiencia $E(n)$ del sistema se mantiene constante y en todo momento por encima de un factor de 0,5. Normalmente todos los sistemas tienen un determinado número de procesadores a partir del cual la eficiencia empieza a disminuir o decaer de forma más o menos brusca. Un sistema es más escalable que otro si este número de procesadores, a partir del cual la eficiencia disminuye, es menor que el otro.

d) Redundancia: En computación paralela se define como la relación $O(n)/O(1)$, que comprende respectivamente el número total de operaciones ejecutadas por el sistema con n procesadores dividido por el número de operaciones ejecutadas cuando utilizamos un solo procesador en la ejecución. Es decir,

$R(n) = O(n)/O(1)$, lo que indica la relación entre el paralelismo en software y en hardware, donde $1 \leq R(n) \leq n$.

e) Utilización: La utilización de un sistema de computación paralela se define como $U(n) = R(n).E(n) = O(n)/nT(n)$. La utilización indica el porcentaje de procesadores, espacio de memoria, ancho de banda de comunicación, y demás recursos que se emplean durante la ejecución de un programa paralelo. AL respecto se puede observar la siguiente relación: $1/n \leq E(n) \leq U(n) \leq 1$ y $1 \leq R(n) \leq 1/E(n) \leq n$.

f) Calidad del paralelismo: La calidad de un sistema paralelo es directamente proporcional al speed-up y la eficiencia, inversamente proporcional a la redundancia. Así, tenemos que:

$$Q(n) = \frac{S(n) * E(n)}{R(n)} = \frac{T^3_{sec}(1)}{n * T^2_{par}(n) * O(n)} \tag{3.5}$$

Dado que $E(n)$ es siempre una fracción y $R(n)$ es un número entre 1 y n, la calidad $Q(n)$ está siempre limitada por $S(n)$ o Speed-Up.

En resumen, usamos el speed-up $S(n)$ para indicar el grado de ganancia de rapidez al usar procesamiento paralelo. La eficiencia $E(n)$ mide la porción útil del trabajo total realizado por n elementos de procesamiento. La redundancia $R(n)$ mide el grado del incremento de la carga sin afectar significativamente el rendimeinto del sistema. La utilización $U(n)$ indica el grado de utilización de recursos durante el procesamiento paralelo. Finalmente, la calidad $Q(n)$ combina el efecto del speed-up, la eficiencia y redundancia en una única expresión para medir el mérito relativo del procesamiento usando un sistema paralelo.

En muchas aplicaciones prácticas, donde es importante la respuesta más rápida posible, la carga de trabajo (Workload) se mantiene fija y es el tiempo de ejecución lo que se debe intentar reducir. Al incrementarse el número de procesadores en el sistema paralelo, la carga de trabajo fija se distribuye entre más procesadores para la ejecución paralela. Por lo tanto, el objetivo principal es obtener los resultados lo más pronto posible. En otras palabras, disminuir el tiempo de respuesta es nuestra principal meta. A la ganancia de tiempo obtenida para este tipo de aplicaciones donde el tiempo de ejecución es crítico se le denomina Speed-Up bajo carga fija.

Al respecto la Ley de Amdahl [Hennessy and Patterson, 2017], formulada por Gene Amdahl, dicta que la mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente.

En el caso de los sistemas de varios elementos de procesamiento (paralelos), este rendimiento está limitado además por la fracción paralela β que se puede explotar (paralelismo inherente al algoritmo de solución del problema). Otra forma de decirlo es que el paralelismo está determinado por la porción secuencial de instrucciones α del algoritmo de solución.

Asumiendo entonces un sistema con n procesadores y un algoritmo o programa con un porcentaje de código no paralelizable o secuencial α, y el resto paralelizable β ($\beta = 1 - \alpha$), se tiene que el rendimiento del sistema se calcula como $S(n) = T(1)/T(n)$. El Speed-Up o $S(n)$ de un programa con un fragmento paralelizado se calcula con:

$$S(n) = \frac{1}{\alpha + \frac{1-\alpha}{n}} = \frac{1}{\alpha + \frac{\beta}{n}} \leq \frac{1}{\alpha} \tag{3.6}$$

3.4.2 *MÉTODOS DE EXPLOTACIÓN DEL PARALELISMO*

Como ya se ha mencionado anteriormente, la computación paralela es una técnica de computación en la que muchas instrucciones asociadas al algorítmo de solución de un problema se ejecutan simultáneamente. Se basa en el principio de que los problemas se pueden dividir en partes más pequeñas que pueden resolverse de forma concurrente (en paralelo).

El paralelismo se ha empleado durante muchos años, sobre todo en la computación de alto rendimiento. Este interés ha crecido en los últimos años debido a las limitaciones físicas que han impidido el aumento de la frecuencia en los microprocesadores secuenciales tradicionales. Esto sumado al hecho que el consumo de energía y la generación de calor de los dispositivos y computadores se han constituido en un objetivo de la eficiencia.

En computación paralela existen dos modelos de programación paralela, uno explicíto y otro implícito. En el **modelo de programación paralela explícita**, el lenguaje de programación debe permitir expresar el algoritmo paralelo de forma taxativa y expresa, y la forma cómo cooperan e interactúan los procesadores y procesos para resolver un problema específico. En este modelo la tarea del compilador del lenguaje es sencilla, en cambio la del programador es bastante difícil porque lo obliga a atender detalles de bajo nivel asociados al paralelismo.

Por otra parte, en el **modelo de programación implícito** se usa un lenguaje de programación secuencial de alto nivel y el compilador inserta las instrucciones necesarias para ejecutar el programa en un computador paralelo, escondiendo de esta manera la expresión explícita del paralelismo. En este caso el compilador tiene que analizar y comprender las dependencias para asegurar una traducción y mapeado eficiente del algoritmo en el computador paralelo.

Por esto, la computación paralela se ha convertido en el paradigma dominante en la arquitectura de computadores y en las herramientas de programación, principalmente en forma de procesadores de múltiples núcleos (multicore) y de muchos núcleos (manycore), así como en APIs de desarrollo de software que aprovecha estos núcleos.

Hay diferentes formas de explotar paralelismo, pero antes se deben revisar algunas definiciones y conceptos necesarios en esta área de la computación.

3.4.2.1 Definiciones básicas en computación:

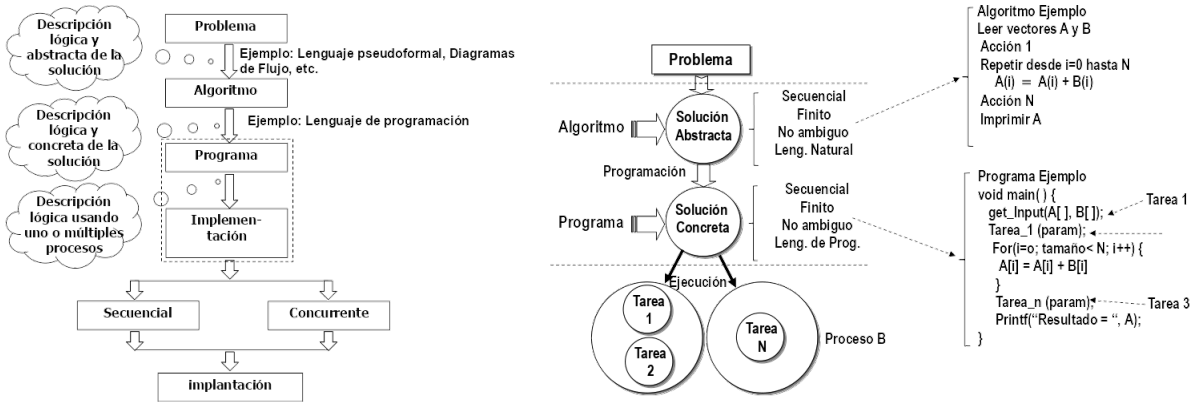
La la noción de **algoritmo** [Gebali, 2011, Acar and Blelloch, 2019] es la base fundamental de las ciencias de la computación. Los algoritmos son producto del estudio de los aspectos teóricos y tecnologicos asociados al proceso abstracto que describe las solución lógica a problemas particulares de interés. Es un proceso sistemático que describe el esquema de tratamiento de datos e información que los transforma en resultados, es decir, es un conjunto finito de pasos o instrucciones que resuelven secuencialmente un problema. Un algoritmo es en definitiva una abstracción de una solución producto del análisis de un problema de modo que separa la lógica de su implementación.

Relacionado con lo anterior, tenemos que un **programa de computación** [Acar and Blelloch, 2019] es la expresión concreta de un algoritmo o solución a un problema, es decir, una descripción lógica orientada a una implementación específica en un computador. No obstante, un programa es una entidad pasiva conformado por un grupo de instrucciones en forma de tareas que describen secuencialmente la solución usando un lenguaje de programación (ver Figura 3.8).

Esta entidad pasiva o **programa de computación** se transforma en una entidad activa denominada **proceso** cuando pasa a un estado de ejecución [Stallings, 2018]. Un **proceso** es entonces un programa ejecutándose en un sistema computacional al cual, por intervención del sistema operativo, le son asignados previamente un conjunto de recursos computacionales (CPU, memoria, pila, registros, dispositivos de entrada/salida, archivos, etc.) necesarios para realizar las intrucciones y tareas que permiten procesar los datos e informacion de entrada y producir los resultados esperados que solucionan el problema.

Conceptualmente, cada **proceso** es en sí mismo un **hilo de ejecución** principal (Main Thread) que es visto como un CPU virtual [Stallings, 2018]. Este proceso pesado o hilo de control principal puede generar otros procesos livianos o hilos dependientes como hilos secundarios o procesos hijos (Child Thread).

Los programas de computación generalmente se estructuran de forma modular dividiendo el programa en **unidades funcionales como subrutinas, procedimientos, funciones u objetos** [Isazadeh et al., 2017]. Esto facilita la legibilidad, la reutilización de estos módulos y la separación de las tareas dentro del programa. En este sentido, una **tarea** es la forma como se agrupan las secciones de instrucciones secuenciales que realizan funciones específicas dentro del programa (ver Figura 3.8).



a) Vista de la relación entre algoritmo, programa, tarea y proceso.

b) Modelo conceptual del algoritmico y su solución en software.

Figura 3.8: a) Jerarquía conceptual de solución computacional a un problema. b) Proyección del problema abstracto a una solución concreta de software. Fuente: Elaborado por el autor.

En el cuerpo de un programa de computación se pueden distinguir dos tipos de secciones de instrucciones, unas donde se realizan tareas o actividades de procesamiento orientadas a gestionar los datos de entrada y de salida o **secciones de código limitadas por datos** (I/O bound); y otras donde se realizan instrucciones de cálculo intensivo sobre esos datos de entrada o **secciones de código limitadas por cómputo** (CPU bound) [Oh, 2017].

En el área de la computación de alto rendimiento son de interés estas secciones limitadas en cómputo dado que en ellas se encuentra el mayor uso del tiempo de CPU debido al procesamiento de los datos. Estas secciones constituyen el núcleo o **Kernel** [Oh, 2017] de procesamiento intensivo sobre grandes dominios de datos.

Por último, la carga de trabajo o **workload** [Pacheco and Malensek, 2020] está determinada por la cantidad de datos a procesar que se le asigna a las tareas asociadas a los procesos en un momento dado.

3.4.2.2 Condiciones de Bernstein y dependencias que limitan el paralelismo:

Un aspecto esencial en la programación paralela es la dependencia de datos lo cual es fundamental en la implementación de algoritmos paralelos. Ningún programa puede ejecutarse más rápido que la cadena más

larga de cálculos dependientes (conocida como la ruta crítica). Esto se debe a que los cálculos que dependen de cálculos previos en la cadena deben ejecutarse en orden o secuencialmente. Sin embargo, la mayoría de los algoritmos no consisten sólo de una larga cadena de cálculos dependientes; generalmente hay oportunidades para ejecutar cálculos independientes en paralelo.

Sea P_i y P_j dos segmentos de un programa. Las condiciones de Bernstein [Bernstein, 1966] describen cuando estos segmentos son independientes y pueden ejecutarse en paralelo. Para P_i, sean I_i todas las variables de entrada y O_i las variables de salida, y del mismo modo para P_j. Ahora, P_i y P_j son independientes sí y solo sí satisfacen las siguientes tres condiciones:

1. $I_j \cap O_i = \varnothing$: Una violación de ésta primera condición introduce una dependencia de flujo, correspondiente al primer segmento que produce un resultado utilizado por el segundo segmento.

2. $I_i \cap O_j = \varnothing$: Esta segunda condición representa una anti-dependencia, cuando el segundo segmento (P_j) produce una variable que necesita el primer segmento (P_i).

3. $O_i \cap O_j = \varnothing$: Esta tercera y última condición representa una dependencia de salida, por ejemplo cuando dos segmentos escriben en el mismo espacio de almacenamiento, el resultado final será el del último segmento ejecutado.

Como un ejemplo, se consideran las siguientes funciones, que demuestran varios tipos de dependencias:

```
1: int func_ConDepend(a, b) {
2:   c = a * b;
3:   d = 3 * c;
4:   return d;
5: }
```

En el código en lenguaje C mostrado arriba, la instrucción de la línea 3 en la función $func\_ConDepend(a, b)$ no puede ejecutarse ni antes ni en paralelo a la instrucción de la línea 2, ya que la instrucción 3 utiliza o depende del resultado de la instrucción de la línea 2. En caso contrario, esto viola la condición 1, y por tanto se presenta como una dependencia de flujo.

En el siguiente ejemplo se muestra un código donde no existen dependencias entre las instrucciones, por lo que todos ellas se pueden ejecutar en paralelo.

```
1: int func_SinDepend(a, b) {
2:   c = a * b;
3:   d = 3 * b;
4:   e = a + b;
5:   return c+d+e;
6: }
```

Las condiciones de Bernstein no permiten que la memoria se comparta entre los diferentes procesos. Por esto son necesarios algunos medios que impongan un ordenamiento entre los accesos tales como semáforos, barreras o algún otro método de sincronización o de exclusión mutua.

3.4.2.3 Grados y niveles de explotación del paralelismo:

Dado que un programa es en esencia una secuencia de instrucciones ejecutadas por un procesador, esto permite que estas instrucciones puedan reordenarse y agruparse para luego ser ejecutadas en paralelo

sin cambiar el resultado del programa. Estos grupos pueden ser de diferentes niveles de granularidad y complejidad para expresar paralelismo [Czarnul, 2018].

En computación paralela la granularidad de una tarea o tamaño del grano de procesamiento es una medida de la cantidad de trabajo (o cálculo) que realiza esa tarea. Otra definición de granularidad se propone en [Czarnul, 2018] donde se tiene en cuenta la sobrecarga de comunicación entre múltiples procesadores o elementos de procesamiento. Entonces, se define la granularidad como la relación entre el tiempo de cálculo y el tiempo de comunicación, donde el tiempo de cálculo es el tiempo necesario para realizar el cálculo de una tarea y el tiempo de comunicación es el tiempo necesario para intercambiar datos entre procesadores.

Si T_{comp} es el tiempo de cálculo y T_{comm} denota el tiempo de comunicación, entonces la Granularidad G de una tarea se puede calcular como $G = T_{comp}/T_{comm}$.

La granularidad generalmente se mide en términos de la cantidad de instrucciones ejecutadas en una tarea o subrutina en particular. Alternativamente, la granularidad también se puede especificar en términos del tiempo de ejecución de un programa, combinando el tiempo de cálculo y el tiempo de comunicación.

Entonces, dependiendo de la cantidad de trabajo realizado por una tarea paralela, el paralelismo se puede clasificar en tres categorías [Magoules et al., 2015]: paralelismo de grano fino, de grano medio y de grano grueso.

En un **paralelismo de grano fino**, un programa se divide en una gran cantidad de pequeñas tareas. Estas tareas se asignan individualmente a muchos procesadores. La cantidad de trabajo asociado con una tarea paralela es baja y el trabajo se distribuye uniformemente entre los procesadores. Por lo tanto, el paralelismo de grano fino facilita el equilibrio de carga de trabajo. Como cada tarea procesa menos datos, la cantidad de procesadores necesarios para realizar el procesamiento completo es alta. Esto, a su vez aumenta la sobrecarga de comunicación y sincronización. Este paralelismo de grano fino se aprovecha mejor en arquitecturas que admiten una comunicación rápida entre elementos simple o básicos de procesamiento. Dado que la arquitectura con memoria compartida tiene una sobrecarga de comunicación baja es la más adecuada para el paralelismo de grano fino.

En el otro extremo tenemos que en el **paralelismo de grano grueso**, un programa se divide en tareas grandes. Debido a esto, se lleva a cabo una gran cantidad de cálculos en los procesadores. Esto puede resultar en un desequilibrio de carga, en el que ciertas tareas procesan la mayor parte de los datos mientras que otras pueden estar inactivas. Además, el paralelismo de grano grueso no aprovecha el paralelismo en el programa, ya que la mayor parte del cálculo se realiza secuencialmente en un procesador. La ventaja de este tipo de paralelismo es la baja sobrecarga de comunicación y sincronización. Dado que la arquitectura con memoria distribuida o paso de mensajes tarda mucho en comunicar datos entre procesos, la hace adecuada para el paralelismo de grano grueso.

Por último, en el **paralelismo de grano medio** se usa en relación con el paralelismo de grano fino y grueso. El paralelismo de grano medio es un compromiso entre el paralelismo de grano fino y el paralelismo de grano grueso, donde tenemos el tamaño de la tarea y el tiempo de comunicación mayor que el paralelismo de grano fino y menor que el paralelismo de grano grueso. La mayoría de los computadores paralelos de uso general entran en esta categoría.

La granularidad está estrechamente relacionada con el nivel de procesamiento paralelo [Magoules et al., 2015]. **Un programa se puede dividir en 4 niveles de paralelismo:** a) Nivel de instrucción, b) Nivel de bucle o ciclo, c) Nivel de tarea o subrutina, y d) Nivel de programa o aplicación.

La mayor cantidad de paralelismo se logra a nivel de instrucción, seguido del paralelismo a nivel de bucle o ciclo. **A nivel de instrucción y de bucle, se logra un paralelismo de grano fino**. El tamaño de grano típico a nivel de instrucción es de 20 instrucciones aproximadamente, mientras que el tamaño de grano a nivel de bucle es de 500 instrucciones. A nivel de tarea, subrutina o procedimiento, el tamaño de grano suele ser de unos pocos miles de instrucciones. **El paralelismo de grano medio se logra a nivel de tarea o subrutina**. **A nivel de programa se utiliza un paralelismo de grano grueso**, tiene lugar la ejecución paralela de programas o aplicaciones. La granularidad puede estar en el rango de decenas de miles de instrucciones.

En conclusión, la granularidad afecta el rendimiento de los sistemas de computación paralelos. El uso de granos finos o tareas pequeñas da como resultado un mayor paralelismo y, por lo tanto, aumenta la aceleración. Sin embargo, la sobrecarga de sincronización, las estrategias de programación, etc. pueden afectar negativamente el desempeño de las tareas de grano fino. El aumento del paralelismo por sí solo no ofrece el mejor rendimiento.

Para reducir la sobrecarga de comunicación, se puede aumentar la granularidad. Las tareas de grano grueso tienen menos sobrecarga de comunicación, pero a menudo causan un desequilibrio de carga. Por lo tanto, se logra un rendimiento óptimo entre los dos extremos del paralelismo de grano fino y de grano grueso. Es necesario hacer un esfuerzo y usar técnicas que ayuden a determinar la mejor granularidad para mejorar el procesamiento paralelo, aunque encontrar el mejor tamaño del grano depende de varios factores y varía mucho de un problema a otro [Magoules et al., 2015].

3.4.2.4 Tipos y formas de explotación del paralelismo:

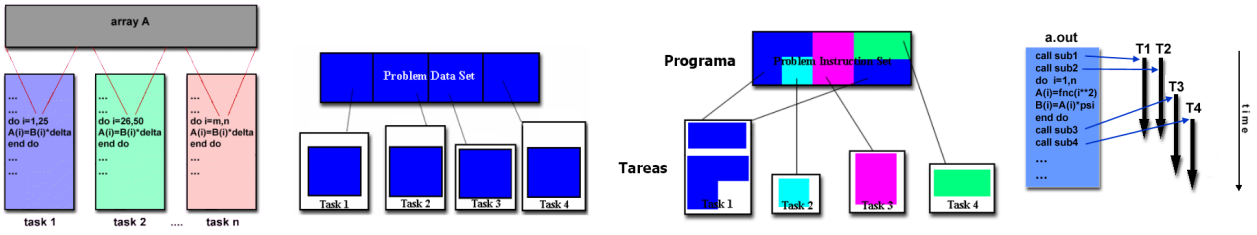
El **grado de paralelismo** es el número de procesos paralelos en los que se puede dividir un programa en un instante dado. La ejecución de un programa de computación paralelo puede utilizar un número diferente de procesadores en diferentes períodos de tiempo. Para cada período de tiempo, existe un número de procesadores que se puede llegar a usar para ejecutar el programa.

En este sentido, las estrategias de explotación del paralelismo buscan determinar la forma mas apropiada o ventajosa de explotar la ejecución simultánea de instrucciones dependiendo del nivel y unidad de agrupación de éstas. Según Pacheco [Pacheco and Malensek, 2020], hay tres formas de explotar paralelismo: **Paralelismo de Datos, Paralelismo de Control o Tareas (aunque se puede usar un híbrido entre ambos), y Paralelismo de Flujo**.

Por un lado, tenemos que el **paralelismo de datos** consiste en aprovechar la paralelización mediante la distribución de los segmentos de datos provenientes de un dominio de datos a través de diferentes elementos de procesamiento que aplican la misma tarea de cómputo sobre estos datos. Se puede aplicar por ejemplo en estructuras de datos regulares como vectores y matrices trabajando sobre cada elemento en paralelo (ver Figura 3.9a).

Por otro lado, el **paralelismo de control, de tareas o de funciones** consiste en la paralelización mediante la distribución de diferentes tareas de cómpuro repartidas por separado para ser realizadas simultáneamente por procesos o subprocesos, en diferentes elemento de procesamiento. A diferencia del paralelismo de datos, que implica ejecutar la misma tarea en diferentes segmentos de datos, el paralelismo de tareas se distingue por ejecutar muchas tareas diferentes al mismo tiempo sobre los mismos datos. Una forma común de paralelismo de tareas es usando un modelo de cómputo paralelo denominado **cause vectorial, segmentación encauzada, canalización o pipeline**, que consiste en mover un solo conjunto de datos

a través de una cadena secuencial de etapas o tareas separadas donde cada etapa o tarea puede ejecutarse independientemente de las demás (ver Figura 3.9b).



a) Ejemplo de Paralelismo de Datos. b) Ejemplo de Paralelismo de Tareas o de Control.

Figura 3.9: a) Ejemplo de Paralelismo de Datos en el procesamiento de un Arreglo o Vector. b) Ejemplo de Paralelismo de Tareas mediante la ejecución de múltiples tareas distintas sobre un dominio de datos. Fuente: https://hpc.llnl.gov/introduction-parallel-computing-tutorial.

En un sistema multiprocesador, el paralelismo se logra cuando cada procesador ejecuta un proceso o subproceso (hilos) igual o diferente sobre los mismos o diferentes datos. Los hilos pueden ejecutar el mismo código o uno diferente. En el caso general, diferentes subprocesos de ejecución se comunican entre sí mientras funcionan, pero esto no es un requisito. La comunicación suele tener lugar pasando datos de un subproceso al siguiente como parte del flujo de trabajo durante el procesamiento.

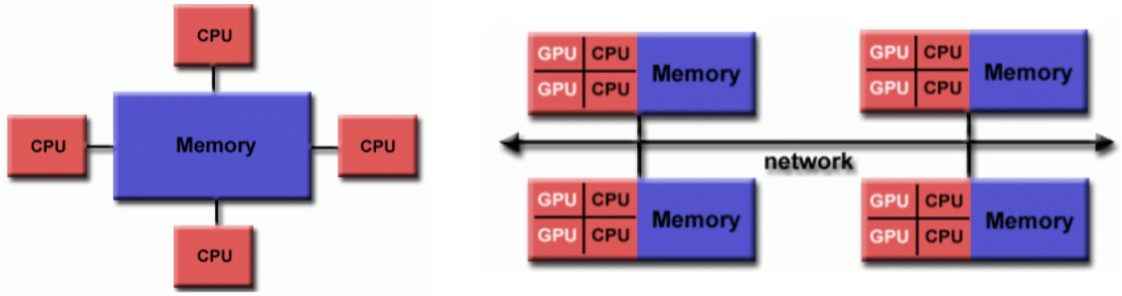
La explotación del paralelismo a nivel de hilos es hoy dia bastante común en los computadores de escritorio debido al auge de los microprocesadores de multiples núcleos en CPU (de grano grueso) y de muchos núcleos en GPU (de grano fino).

En definitiva, el paralelismo de tareas enfatiza la naturaleza distribuida de las tareas de procesamiento, a diferencia del paralelismo de datos que enfatiza la naturaleza distribuida de los datos a procesar. La mayoría de las soluciones a los problemas del mundo real caen en algún lugar entre el paralelismo de tareas y el paralelismo de datos.

Un último aspecto que se desea abordar en la presente sección esta asociado a la forma como se organiza el modelo de memoria en la computación paralela actual [Schmidt et al., 2018]. **Según el modelo de memoria, existen dos formas: la computación paralela con memoria compartida y la computación paralela con memoria distribuída**, pero también es posible un híbrido de ambas.

En la **computación con memoria compartida** tenemos un computador con varios elementos de procesamiento y memorias conectados a tráves de un mismo bus interno del sistema fuertemente acoplado. En este escenario, los procesadores comparten una misma memoria principal, es decir, para todos es visible el mismo espacio común de direcciones (ver Figura 3.10a).

Por otra parte, en la **computación con memoria distribuida** tenemos varios computadores autónomos con su propio cpu, memoria, disco, dispositivos de entrada/salida, etc., conectados en red que se presentan al usuario como un solo sistema. Por tanto, en los sistemas con memoria distribuida no hay memoria compartida o común, es decir, dado que cada computador posee su propia memoria principal este espacio de direcciones es invisible a los otros computadores, y se comunican entre sí a través del paso de mensajes para compartir datos (ver Figura 3.10b).



a) Ejemplo de Sistema con Memoria Compartida. a) Ejemplo de Sistema con Memoria Distribuida.

Figura 3.10: a) Ejemplo de Sistema con Memoria Compartida. b) Ejemplo de Sistema Heterogéneo con Memoria Distribuida. Fuente: https://hpc.llnl.gov/introduction-parallel-computing-tutorial.

3.4.3 *ALGUNOS ALGORITMOS USADOS EN COMPUTACIÓN PARALELA*

En la figura 3.11 se muestra la representación gráfica de algunos modelos de algortimos para computación paralela que sirven de patrones para diseñar esqueletos algorítmicos. Entre los modelos de algortimos comúnmente usados, encontrados en [Fischer et al., 2003] y [McCool et al., 2012], tenemos:

3.4.3.1 Granja de Tareas (Tasks Farm) o Maestro/Esclavo (Master/Slave):

En este modelo uno o más procesos gestores o maestros generan trabajo o tareas y lo asignan o reparten entre varios procesos trabajadores o esclavos. Las tareas pueden asignarse a priori de forma estática siempre que el gestor pueda estimar el tamaño de las tareas o, hacer una asignación aleatoria para lograr un equilibrio en la distribución de la carga de trabajo.

En algunos casos, puede ser necesario realizar el trabajo por fases, debiendo el trabajo de cada fase terminar antes de que se pueda generar el trabajo de las fases siguientes. En este caso, el gestor puede hacer que todos los trabajadores se sincronicen después de cada fase (barrera de sincronización).

El modelo gestor/trabajador puede generalizarse a un modelo jerárquico o multinivel gestor-trabajador, en el que el gestor o maestro de nivel superior distribuye las tareas a los gestores de segundo nivel, que a su vez subdividen estas tareas entre sus propios trabajadores, pudiendo ellos mismo también hacer trabajo. En general, este modelo es adecuado para los paradigmas de memoria compartida o de memoria distribuida, ya que la interacción es bidireccional; es decir, el gestor reparte trabajo a los trabajadores, y los trabajadores reciben trabajo del gestor.

3.4.3.2 Segmentación Encauzada (Pipeline):

En el modelo de encauzamiento o pipeline un flujo de datos pasa a través de una cadena o secuencia de etapas o procesos, cada uno de los cuales realiza alguna tarea sobre esos datos. En este modelo cada etapa o proceso realiza una tarea distinta, las cuales se ejecutan simultáneamente sobre el flujo de datos que atraviesa el pipeline. Esto también se denomina paralelismo de flujos, dado que existe una cadena lineal de productores y consumidores. Cada etapa o proceso de la cadena puede considerarse consumidor de los datos que le entrega el proceso o etapa que le precede en la cadena y productor de los datos que entrega al proceso o etapa que le sigue en la cadena.

3.4.3.3 Mapear/Reducir (Map and Reduce):

Es un modelo combinado de funciones para procesar y generar grandes conjuntos de datos. El modelo **Map** toma como entrada una función y un conjunto de valores o datos de entrada. A continuación, aplica la

función a cada valor del conjunto de datos y genera otro subconjunto de datos modelados en tuplas o pares (clave/valor). Por otro lado, el modelo **Reduce** es la funcion (complementaria y posterior a la aplicación de la función **Map**) que combina y reduce el conjunto de datos entregados por **Map** utilizando para ello una operación binaria. Por ejemplo, puede utilizar "+" para sumar todos los elementos del conjunto o secuencia de datos.

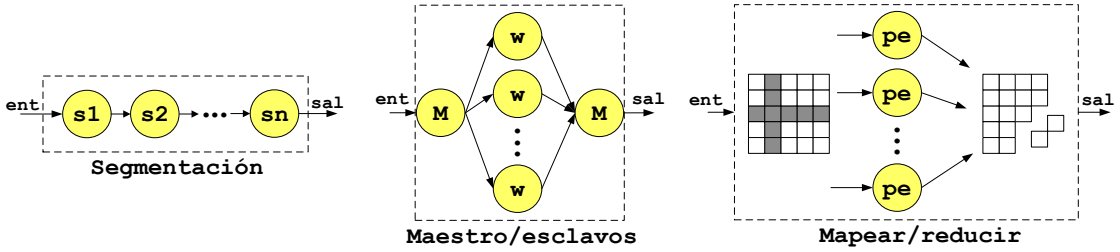


Figura 3.11: Algunos Modelos de Algoritmos para Cómputo Paralelo (Esqueletos Algorítmicos). Fuente: Elaborado por el autor.

3.4.3.4 Ejemplo práctico de programación paralela usando un lenguaje de alto nivel

Como una manera de demostrar la aplicación de los conceptos asociados a los aspectos teóricos ya explicados hasta aquí se utiliza un ejemplo práctico y simple donde se resaltan las ventajas del procesamiento paralelo con respecto a la computación secuencial.

El programa fuente usado como ejemplo se puede observar en el Listado 3.1. Aquí se muestra un programa paralelo construido en base al API openMP/C++ donde se resuelve el problema de elevar al cuadrado x^2 cada elemento de una larga lista de números enteros de entrada, resuelto con una solución secuencial y luego con una solución paralela.

El código fuente mostrado en 3.1 es totalmente funcional. Se ha codificado usando el editor de texto **gedit**, compilado y ejecutado en un computador con CPU IntelCore i7 de 4 núcleos con 4GB de memoria RAM. Todo esto usando como sistema operativo la distribucion Linux Ubuntu LTS 20.04. A continuación se describe el proceso previo de preparación del ambiente mediante la instalación del compilador **GNU g++** (C++) y de las librerías del API openMP versión 1.2:

```
$> sudo apt update; apt upgrade
$> sudo apt install build-essential -y
$> sudo apt-get install g++
$> sudo apt-get install libomp-dev
```

Luego, se compila y corre el programa ejemplo obteniendo el siguiente resultado:

```
$> g++ -fopenmp -lgomp ./openmp-ejemplo.cpp -o openmp-ejemplo
$> ./openmp-ejemplo
$> Este computador tiene 4 núcleo(s)
   Se usaron 5 hilos para procesamiento paralelo
   Se ejecutaron 10 corridas del programa con vectores desde 1.000 hasta 10.000 datos.
   Tabla (x, y1, y2, y3):
   Workload (datos) TSecuencial   TParalelo   SpeedUp:
   x                y1(segs)      y2(segs)    y3
   0                0.0           0.0     0
   1000             0.115870      0.156141    0,74
   2000             0.239220      0.094140    2,54
   3000             0.357060      0.125600    3,08
   4000             0.482070      0.158930    3,03
   5000             0.600300      0.192060    3,12
   6000             0.705280      0.227040    3,10
   7000             0.836380      0.264760    3,15
   8000             0.960150      0.303770    3,16
   9000             1.074280      0.332150    3,23
   10000            1.198280      0.365640    3,27
   --->Se ha creado exitosamente el archivo: "tabla_tiempos-sec-par.dat".
$>
```

Los resultados obtenidos presentan la medición y posterior comparación de los tiempos de ejecución secueniales y paralelos del programa ejemplo usado. En estos se puede notar que para listas de datos de dimensiones pequeñas la solución secuencial con un (1) proceso es más rápida y eficiente, pero pronto a partir de un punto a medida que la dimensión de las listas de datos aumenta, usando cinco (5) hilos o procesos simultáneos, se observa que los tiempos de ejecución paralela son mejores y más eficientes que el secuencial, como se puede ver en las gráficas de la Figura 3.12.

```
/*#########################################################################################
# Autor: Carlos Acosta (25/02/2022)
# Nombre del archivo: openmp-ejemplo3.cpp
# Función: Ejemplo en openMP/C++ que muestra una solución secuencial y paralela.
#########################################################################################*/
#include <stdio.h>
#include <iostream>
#include <omp.h> // Librería de openMP
#include <ctime>
#include <time.h>
using std::cout;
using std::endl;

// Declaración de variables globales
#define BILLION  1000000000.0
const int num_MAXIMO  = 10;   // Valor máximo de números generados aleatoriamente
const int ePrecision  = 1000; // 10^3 Reduce cant. de cifras significativas para tabular

int main(){
    // Declaración de funciones globales
    void genDatos(double entVector[], long int tam_list, int numMAX);
    void tarea(double entVector[], long int tamVectorDatos, double outVector[]);
    // Declaración de variables locales
    FILE *archivoSal;
    const char EndOfLine[]  = "\n";
    int numHilos            = 10;
    long int tamVectorDatos = 1000;// Tamaño inicial del Vector de datos
    int numCorridas         = 10;
```

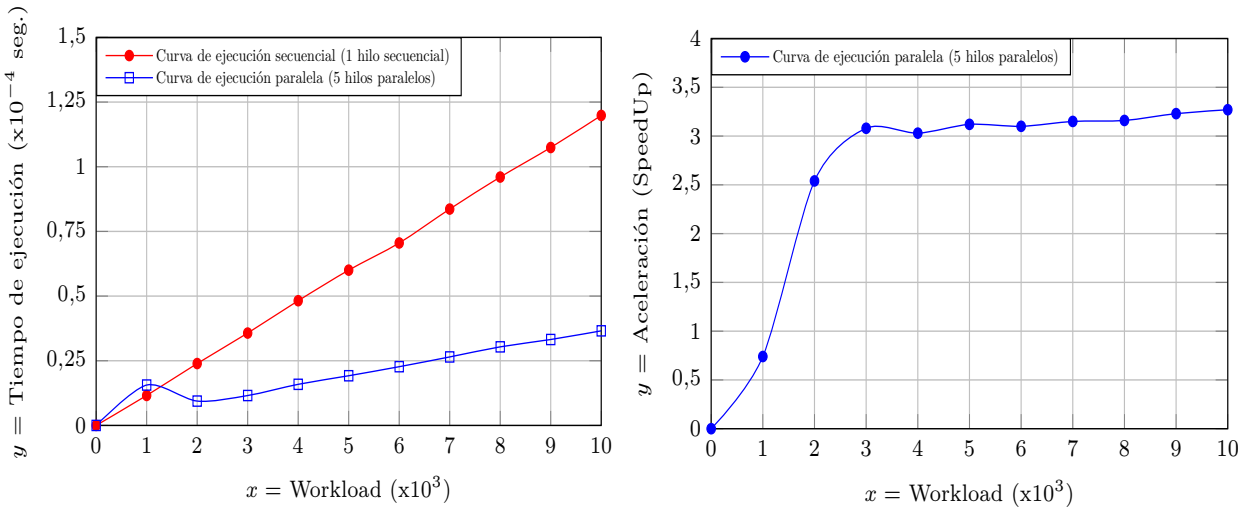
Código fuente 3.1: Ejemplo de programa funcional en openMP/C++ donde se muestra una solución secuencial y otra paralela para resolver un mismo problema. Fuente: Elaborado por el autor.

```
29      int numColTabla         = 4; // Columnas para Tamaño - Tsec - Tpar - Acel
30      double tabla_tiempos[numCorridas][numColTabla]; // arr_tiempos[filas][columnas]
31      struct timespec init_time, end_time;
32      double total_time       = 0;
33      int numProcesadores     = omp_get_num_procs();
34      char nombreArchivoSal[] = "tabla_tiempos-sec-par-1.dat";
35
36      archivoSal = fopen("tabla_tiempos-sec-par.dat", "w+"); // Nombre FILE y modo escritura: "w"
37      if(archivoSal==NULL) {
38          printf("Error creando el archivo de salida %s:\n", nombreArchivoSal);
39          exit(1);
40      }
41      /* SE HACEN UNA O VARIAS CORRIDAS */
42      int pasoMuestra   = tamVectorDatos;
43      for(int i=0; i<numCorridas; i++){
44          // Valor inicial del tamaño de la muestra
45          tamVectorDatos = (i+1)*pasoMuestra;
46          double* inVector = new double[tamVectorDatos];// Se reserva espacio de memoria para los vectores
47          double* outVector = new double[tamVectorDatos];
48          // ENTRADA DE DATOS DEL PROGRAMA
49          tabla_tiempos[i][0] = tamVectorDatos; // Colectando el tamaño Vector de Datos
50          //---> Generando Lista de datos a procesar\n");
51          genDatos(inVector, tamVectorDatos, num_MAXIMO);
52          // ZONA SECUENCIAL
53          /* Ejecutando la Tarea secuencial numHilos veces para procesamiento */
54          clock_gettime(CLOCK_REALTIME, &init_time); // Hora de inicio del procesamiento
55          for(int i=0; i<numHilos; ++i) {
56              tarea(inVector, tamVectorDatos, outVector);
57          }
58          clock_gettime(CLOCK_REALTIME, &end_time);  // Hora de final del procesamiento
59          /* Almacenando tiempo sec en la tabla de tiempos */
60          total_time = (end_time.tv_sec - init_time.tv_sec) + ((end_time.tv_nsec
61                                      - init_time.tv_nsec)/BILLION);
62          tabla_tiempos[i][1] = total_time; // Colectando tiempo secuencial
63          // ZONA PARALELA
64          /* Se obtiene la cantidad de procesadores disponibles en el computador */
65          omp_set_num_threads(numHilos);
66          /* Ejecutando la Tarea paralela con "numHilos" paralelos para procesamiento */
67          clock_gettime(CLOCK_REALTIME, &init_time); // Hora de inicio del procesamiento
68          #pragma omp parallel num_threads(numHilos)
69          {
70              /* Se obtiene el ID de cada hilo paralelo */
71              int idHilo = omp_get_thread_num();
72              tarea(inVector, tamVectorDatos, outVector);
73          }
74          clock_gettime(CLOCK_REALTIME, &end_time); // Hora de final del procesamiento
75          /* Almacenando tiempo paralelo en la tabla de tiempos */
76          total_time = (end_time.tv_sec - init_time.tv_sec) +
77                          ((end_time.tv_nsec-init_time.tv_nsec)/BILLION);
78          tabla_tiempos[i][2] = total_time; // Colectando tiempo paralelo
79          tabla_tiempos[i][3] = tabla_tiempos[i][1]/tabla_tiempos[i][2]; // Calculando SpeedUp
80      }/* FIN FOR */
81      printf("Este computador tiene %d núcleo(s)\n", numProcesadores);
82      printf("Se usaron %d hilos para procesamiento paralelo\n", numHilos);
83      printf("Se ejecutaron %d corridas del programa\n", numCorridas);
84      // Se muestra en pantalla la Tabla de Tiempos
85      printf("Tabla:\nTamaño Lista TSecuencial  TParalelo       SpeedUp\n");
86      printf("x             y1(seg)       y2(seg)       y3\n");
87      printf("%d             %f      %f,     %f\n", 0, 0.0, 0.0, 0.0);
88      for(int i=0; i<numCorridas; i++){
89          printf("%d           %f     %f      %f\n", int(tabla_tiempos[i][0]), tabla_tiempos[i][1],
90                      tabla_tiempos[i][2], tabla_tiempos[i][3]);
```

(Cont. Codigo fuente 3.1) Ejemplo de programa funcional en openMP/C++ donde se muestra una solución secuencial y otra paralela para resolver un mismo problema. Fuente: Elaborado por el autor.

```
91      }
92      // CREANDO ARCHIVO CON TABLA DE TIEMPOS
93      // Grabando cabecera de las columnas de datos
94      fprintf(archivoSal,"%s", "%Cant. Datos | TParalelo (y1) | TSecuencial (y2) | SpeedUp(y3)\n");
95      fprintf(archivoSal,"%s", "x        y1              y2              y3\n");
96      fprintf(archivoSal,"%s", "0        0.0             0.0             0.0\n");
97      // Grabando los tiempos de procesamiento de la corrida
98      for(int i=0; i<numCorridas; i++){
99          for(int j=0; j<numColTabla; j++){
100             if(j==0) fprintf(archivoSal,"%d        ", int(tabla_tiempos[i][j])/pasoMuestra);//Tam Datos
101             if(j<numColTabla-2)  fprintf(archivoSal,"%f         ", tabla_tiempos[i][j+1]*ePrecision);
102             if(j==numColTabla-1) fprintf(archivoSal,"%f         ", tabla_tiempos[i][j]);//SpeedUp
103         }
104         fprintf(archivoSal,"\n");
105     }
106     fclose(archivoSal); // Cerrando el archivo de salida
107     printf("--->Se ha creado exitosamente el archivo:\n %s con la tabla de tiempos\n",
108                     nombreArchivoSal);
109 return 0; /* Programa Termina NORMAL */
110 }
111 /************************************************/
112 // Definición de funciones
113 /* Función que calcula el cuadrado de cada número de un Vector de datos */
114 void tarea(double entVector[], long int tam_list, double outVector[]) {
115     for(long int i=0; i<tam_list; ++i)
116         outVector[i] = entVector[i]*entVector[i];
117 }
118 /* Función que genera un Vector de números aleatorios de tamaño N */
119 void genDatos(double entVector[], long int tam_list, int numMAX) {
120     srand(time(0));
121     for(long int i=0; i<tam_list; ++i)
122         entVector[i] = rand()%numMAX;
123 }
```

(Cont. Codigo fuente 3.1) Ejemplo de programa funcional en openMP/C++ donde se muestra una solución secuencial y otra paralela para resolver un mismo problema. Fuente: Elaborado por el autor.



(a) Curvas de Tiempos de ejecución

(b) Aceleración (SpeedUp) con 5 hilos paralelos

Figura 3.12: Gráfico que muestra las curvas de tiempos de ejecución secuencial con 1 proceso o hilo versus la ejecución paralela con 5 procesos o hilos simultáneos resultado de la ejecución del programa ejemplo en openMP/C++ (Ver código fuente 3.1). Fuente: Elaborado por el autor.

3.4.4 *ESQUELETOS ALGORÍTMICOS Y FUNCIONES DE ORDEN SUPERIOR*

Existen técnicas de abstracción de alto nivel usadas en la programación paralela, como las descritas en [Czarnul, 2018] y [Dubey, 2009], que pueden ser apropiadas para desarrollar aplicaciones paralelas embebidas de alto rendimiento sobre sistemas de computación reconfigurable usando un enfoque hardware/software. Paradigmas como el Orientado a Objetos [Booch et al., 2007], Patrones de Diseño [Gamma et al., 1998] y especialmente los Esqueletos Algorítmicos de Cole ("Cole's Algorithmic Skeletons") [Cole, 2004b] han sido herramientas de alto nivel exitosas en la programación de software paralelo.

3.4.4.1 Definición y características de los Esqueletos Algorítmicos:

El paradigma de Esqueletos Algorítmicos (Skeletal Programming o Algorithmic Skeletons), representa en la presente investigación un recurso clave en la propuesta de solución. Este paradigma fue propuesto por el Profesor Murray Cole [Cole, 1989b] del "Institute for Computing Systems Architecture (ICSA)" de la Universidad de Edimburgo, Escocia-Reino Unido. Introdujo los Esqueletos Algorítmicos (Algorithmic Skeletons) como un mecanismo de encapsulamiento del paralelismo. Son una forma estructurada de abstracción que permite separar un patrón de computación paralela de su implementación.

Los Esqueletos Algorítmicos son bloques básicos de construcción de programas genéricos, portátiles y reutilizables para los que pueden estar disponibles implementaciones paralelas. Estos esconden al programador de una aplicación los detalles de una implementación paralela, proporcionándole una implementación eficiente de una solución paralela apropiada para un problema específico.

3.4.4.2 Funciones de orden superior y sus propiedades:

Siguiendo el razonamiento anterior, es oportuno resaltar que los esqueletos algoritmicos pueden ser vistos también como funciones de orden superior (higher order functions). Esta clase de funciones posee un conjunto de propiedades que son útiles para el diseño e implementación de esqueletos algorítmicos, como se obeserva en la Figura 3.13. Además, este enfoque permite la programación de aplicaciones combinando y componiendo funciones secuencialmente como en la programación funcional [Tran, 2022], para escribir programas que exploten paralelismo de forma implícita, portable e independientes de la arquitectura paralela.

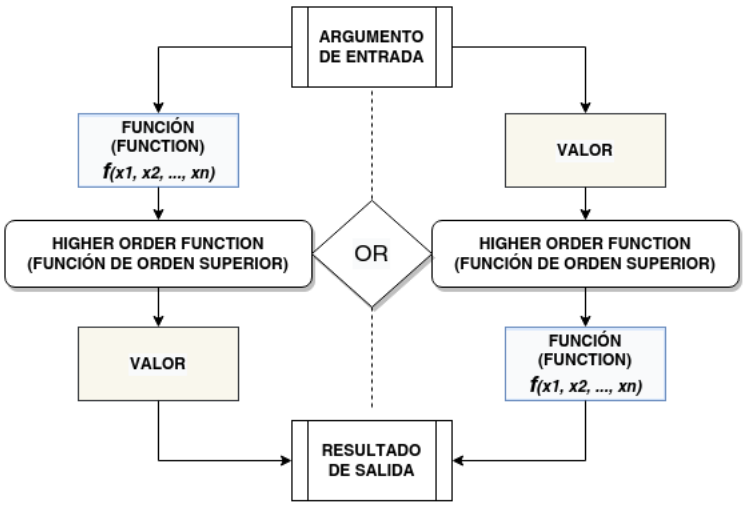


Figura 3.13: Propiedades de las funciones de orden superior como mecanismo de diseño de esqueletos algorítmicos. Fuente: Elaborado por el autor.

Como ejemplo, usamos el lenguaje de programación Python [Francis, 2021], el cual provee recursos de alto nivel para implementar funciones de orden superior. Para mostrar esto se han codificado ejemplos usando el editor de texto **gedit**, ejecutados en un computador con CPU IntelCore i7 de 4 núcleos físicos con 4GB de memoria RAM. Todo esto usando como sistema operativo la distribucion Linux Ubuntu LTS 20.04. A continuación se realiza un proceso previo de preparación del ambiente para correr los ejemplos mediante la instalación del intérprete Python version 3.0:

```
$> sudo apt update; apt -y upgrade
$> sudo apt install -y python3-pip
```

Entonces, para entender cada una de las propiedades de las funciones de orden superior se muestran ejemplos con programas en Python:

a) Propiedad 1: Función asignada como objeto o variable de otra función. Se puede almacenar una función dentro de una variable. En Python, una función puede ser asignada a una variable. Aquí se crea una referencia a la función, como se muestra en el código fuente de ejemplo 3.2:

```
1  ##############################################################################################
2  # Autor: Carlos Acosta (25/02/2022)
3  # Ejemplo Python: Función f(x) asignada a una variable de tipo función f(y).
4  ##############################################################################################
5  # Definición de la función f(x): SUMA dos números enteros
6  def funcion_x(valor1, valor2):
7      return valor1 + valor2
8  # Lectura de valores
9  valor1 = int(input("Ingrese primer  número entero: "))
10 valor2 = int(input("Ingrese segundo número entero: "))
11 # Resultado de la función f(x)
12 print('Resultado de f(x): ', funcion_x(valor1, valor2))
13 # Asinación de la función f(x) a una variable f(y) de tipo objeto función
14 funcion_y = funcion_x # <--- AQUI SE ASIGNA UNA FUNCION COMO UNA VARIABLE U OBJETO
15 # Resultado de la función f(y)
16 print('Resultado de f(y): ', funcion_y(valor1, valor2))
```

Código fuente 3.2: Ejemplo de programa en Python de la propiedad 1 de las funciones de orden superior donde se muestra una función asignada a una variable. Fuente: Elaborado por el autor.

Al ejecutar el código fuente mostrado en 3.2 se produce la siguiente salida:

```
$> python3 codigo-funcionComoObjeto.py
   Ingrese primer  numero entero: 2
   Ingrese segundo numero entero: 5
   Resultado de f(x):  7
   Resultado de f(y):  7
```

b) Propiedad 2: Función que retorna como resultado otra función. Una función puede devolver como resultado a otra función o actuar como instancia de un objeto. Como las funciones son objetos, también podemos devolver una función desde otra función. A continuación se muestra un ejemplo en el código fuente 3.3:

```python
#################################################################################################
# Autor: Carlos Acosta (25/02/2022)
# Ejemplo Python: Función f(x) que retorna otra función f(y) como resultado.
#################################################################################################
# Definición de la función f(x): SUMA dos números enteros
def funcion_x(valor1):
    # Función f(y) anidada para un segundo valor
    def funcion_y(valor2):
        return valor1 + valor2 # Suma los dos valores
    return funcion_y # Se retorna la función f(y) como resultado
# Lectura de valores
valor1 = int(input("Ingrese primer  número entero: "))
valor2 = int(input("Ingrese segundo número entero: "))
# Se asina la función f(x) anidada con f(y) a una variable x tipo objeto
var_x = funcion_x(valor1) # <--- AQUI SE RETORNA UNA FUNCION COMO RESULTADO DE OTRA FUNCION
# Se usa una variable tipo objeto como una función de orden superior
var_y = var_x(valor2) <--- AQUI SE USA LA FUNCION RETORNADA
# Resultado de la composición de funciones f(x, f(y))
print('Resultado de f(x, f(y)): ', var_y)
```

Código fuente 3.3: Ejemplo de programa en Python de la propiedad 2 de las funciones de orden superior donde se muestra una función que devuelve otra función como resultado. Fuente: Elaborado por el autor.

Con la ejecución del anterior código fuente 3.3 se obtiene la siguiente salida:

```
$> python3 codigo-funcionRetornafuncion.py
   Ingrese primer  numero entero: 1
   Ingrese segundo numero entero: 2
   Resultado de f(f(y)): 3
```

c) Propiedad 3: Función que toma como argumento a otra función. Podemos pasar una función como argumento a otra función. Las funciones son como objetos en Python, por lo que pueden pasarse como argumento a otra función. A continuación se muestra un ejemplo en el código fuente 3.4:

```python
#################################################################################################
# Autor: Carlos Acosta (25/02/2022)
# Ejemplo Python: Función f(x) que toma como argumento o parámetro a otra función f(y).
#################################################################################################
# Definición de la función f(x) que SUMA dos números enteros
def funcion_x(valor1, valor2):
    return valor1 + valor2 # Suma los dos valores
# Definición de la función genérica f(y) que toma como parámetros
# a la función f(x) y sus argumentos
def funcion_gen(funcion, valor1, valor2):
    return funcion(valor1, valor2) # Aplica la función pasada como
                                   # parámetro a sus argumentos
# Lectura de valores
valor1 = int(input("Ingrese primer  número entero: "))
valor2 = int(input("Ingrese segundo número entero: "))
Resultado = funcion_gen(funcion_x, valor1, valor2) # <--- AQUI SE PASA UNA FUNCION
                                                   #      COMO ARGUMENTO DE OTRA FUNCION
# Resultado de la composición de funciones f(x, f(y))
print('Resultado de f(x, f(y)): ', Resultado)
```

Código fuente 3.4: Ejemplo de programa en Python de la propiedad 3 de las funciones de orden superior donde se puede observar a una función que toma como parámetro o argumento a otra función. Fuente: Elaborado por el autor.

El resultado de salida al correr el código fuente mostrado en 3.4 es el siguiente:

```
$> python3 codigo-funcionComoArgumento.py
   Ingrese primer  numero entero: 7
```

```
Ingrese segundo numero entero: 4
Resultado de f(x, f(y)): 11
```

3.4.5 *EJEMPLO PRÁCTICO DE PROGRAMACIÓN PARALELA IMPLÍCITA USANDO ESQUELETOS ALGORITMICOS COMO FUNCIONES DE ORDEN SUPERIOR*

Como ya se mencionó anteriormente, los esqueletos algorítmicos pueden ser expresados como funciones de orden superior (higher order functions). A manera de ejemplo, se puede mostrar la aplicación de las propiedades de las funciones orden superior incorporadas en un lenguaje de programación de alto nivel. Para ello, se ha seleccionado el lenguaje de programación Python para especificar las versiones secuenciales de algunos de los algoritmos paralelos descritos anteriormente. Particularmente, se muestran los códigos fuente 3.5 y 3.6 en Python con ejemplos de implementación secuencial de las funciones **map**[ ] {**map(operador)[lista]**} y **reduce**[ ] {**reduce(operador)[lista]**}, donde se presentan los resultados esperados de la ejecución de dichos programas.

```python
#################################################################################################
# Autor: Carlos Acosta (25/02/2022)
# Ejemplo Python: Modelo de Cómputo MAP paralelo.
#################################################################################################
# Librería para programación paralela en Python
from multiprocessing import Pool
import time

# Definición de la funcion f(x) que eleva al cuadrado ^2 los valores de un Vector de entrada.
def f(x):
    return x*x # ^2

# Definición del proceso genérico que toma la función f(x) como argumento y sus parámetros
def proceso(funcion, valor1, valor2):
    return funcion(valor1, valor2) # Aplica la función pasada como parámetro a su argumentos

# Creando vector de datos y resultados
VectorEntrada = []
VectorSalida  = []

# Lectura de lista de elementos de entrada
print("Ingrese lista de 4 valores enteros:")
for i in range(4):
     valor = int(input())
     VectorEntrada.append(valor,)

# Muestra en pantalla la lista de entrada
print(f"Lista de Entrada --> {VectorEntrada}")

# Se crea y ejecuta proceso MAP paralelo que toma a f(x) como argumento
print("Creando y ejecutando procesos map paralelos...")
inicio = time.time() # Registra el inicio del procesamiento paralelo
if __name__ == '__main__':
    with Pool(5) as p:
        VectorSalida = list(proceso(p.map, f, VectorEntrada))
fin = time.time() # Registra el fin del procesamiento paralelo

# Muestra resultado y tiempo de procesamiento de la funcion MAP
print(f"Lista de Salida  --> map(^2){VectorEntrada} = {VectorSalida}, en", fin-inicio, "segundos")
```

Código fuente 3.5: Ejemplo de programa en Python donde se muestra el uso de la propiedad 3 de funciones de orden superior para implementar una versión secuencial de la función $map(operador)[e_1, e_2, e_3, ..., e_n]$.
Fuente: Elaborado por el autor.

Cuando se ejecuta el programa del modelo de cómputo **map** mostrado en 3.5 se produce la siguiente salida:

```
$> python3 codigo-funcionMAP.py
   Ingrese un Vector de 4 valores enteros:
   Vector de Entrada --> [-2, 4, -6, 8]
   Vector de Salida  --> map(^2)[-2, 4, -6, 8] = [4, 16, 36, 64]
```

También, en el código fuente 3.6 se muestra un ejemplo de implementación secuencial en Python de la función **reduce**.

```
1  ##############################################################################################
2  # Autor: Carlos Acosta (25/02/2022)
3  # Ejemplo Python: Función 'reduce[]' aplica un operador acumulando el resultado de
4  #                 los elementos de lista de datos de entrada.
5  ##############################################################################################
6  from functools import reduce
7  import time
8
9  # Definición de la funcion f(x) que eleva al cuadrado "^2" los valores de entrada.
10 def operador_cuadrado(elemento):
11     return elemento ** 2 # eleva elemento al cuadrado "^2" = elemento^2
12 # Definición de la función generica f(y) toma como argumento la función f(x) y sus parámetros
13 def funcion_gen(funcion, valor1, valor2):
14     return funcion(valor1, valor2) # Aplica la función pasada como parámetro a su argumentos
15 # Creando listas de datos y resultados
16 VectorEntrada = []
17 # Lectura de lista de elementos de entrada
18 print("Ingrese lista de 4 valores enteros:")
19 for i in range(4):
20      valor = int(input())
21      VectorEntrada.append(valor,)
22 print(f"Lista de Entrada       --> {VectorEntrada}")
23 # Se aplica a la lista de entrada la función f(x) que se pasa como argumento
24 inicio = time.time() # Registra el inicio del procesamiento paralelo
25 suma = funcion_gen(reduce, lambda x, y: x + y, VectorEntrada)
26 final = time.time() # Registra el inicio del procesamiento paralelo
27 # Muestra resultado de la funcion MAP
28 print(f"Valor de la reducción  --> reduce(+){VectorEntrada} =",\
29                                     suma, "en", final-inicio, "segundos")
```

Código fuente 3.6: Ejemplo de programa en Python donde se muestra el uso de la propiedad 3 de funciones de orden superior para implementar una versión secuencial de la función $reduce(operador)[e_1, e_2, e_3, ..., e_n]$. Fuente: Elaborado por el autor.

La ejecución del programa del modelo de cómputo **reduce** mostrado en el código fuente 3.4 genera la siguiente salida:

```
$> python3 codigo-funcionREDUCE.py
   Ingrese un Vector de 4 valores enteros:
   Vector de Entrada     --> [-2, 4, -6, 8]
   Valor de la reducción --> reduce(+)[-2, 4, -6, 8] = 4
```

Por últmo es oportuno mencionar que el conjunto adecuado de esqueletos, su grado de compatibilidad, generalidad e independencia de la arquitectura y las mejores formas de incorporarlos a los lenguajes de programación se han estado investigando intensamente desde los años sesenta hasta la actualidad.

3.5 OPENCL: LENGUAJE ABIERTO PARA LA PROG. DE SISTEMAS HETEROGÉNEOS

El lenguaje OpenCL (OCL, Open Computing Language) fue creado originalmente por Apple quien luego delegó al Grupo Khronos [Khronos OpenCL Working Group, 2023] para convertirlo en un estándar abierto

y libre que no dependa del hardware de un determinado fabricante (CUDA sólo está disponible en tarjetas gráficas NVidia y Stream en tarjetas gráficas de ATI).

OpenCL es el primer lenguaje de programación estándar abierto, gratuito y multiplataforma de alto nivel, basado en la sintaxis convencional del lenguaje C (ANSI C), para la programación paralela orientada a sistemas de cómputo heterogéneo, los cuales actualmente pueden incorporar al menos dispositivos de procesamiento como CPUs, GPUs y FPGAs.

Sin embargo, la característica más importante de la programación con OpenCL [Kaeli et al., 2015] es la capacidad de expresar de forma eficiente construcciones paralelas. El lenguaje permite construcciones para secciones paralelas y expresiónes para comunicación entre procesos paralelos. Además, este lenguaje posee extensiones que permiten aprovechar las características del hardware y describir algorítmicamente tareas sin atender a complejos detalles asociados al hardware donde se va a implementar.

Puesto que es un lenguaje de alto nivel es posible convertir algoritmos de software en implementaciones en hardware con mucha facilidad, dado que es independiente del hardware sobre el cual se va a realizar la implementación [Waidyasooriya et al., 2018]. De esta forma el diseño de programas puede descomponerse en bloques modulares independientes. Esto permite el diseño de sistemas integrados, bajo un enfoque software/hardware y no como dos aspectos separados. También, esto permite que el hardware se construya de la misma forma que el software y pueda ser probado y depurado de manera similar y fácilmente. Así, es posible compilar el mismo código fuente o programa en diferentes plataformas con CPU, GPU y FPGA.

3.5.1 *MODELO DE PLATAFORMA (PLATFORM MODEL) DE OPENCL*

El modelo de plataforma de OpenCL [Khronos OpenCL Working Group, 2023] es una abstracción que describe cómo OpenCL visualiza el hardware. La relación entre los elementos del modelo de plataforma y el hardware en un sistema heterogéneo puede ser una propiedad fija asociada a un dispositivo de cómputo o puede ser una característica dinámica asociada a un programa que depende de cómo un compilador optimiza el código para utilizar mejor el hardware físico.

El modelo consiste en un Host o Anfitrón conectado a uno o más dispositivos de cómputo OpenCL (CPUs, GPUs, FPGAs, DSPs, etc.). Un dispositivo OpenCL se divide en una o más unidades de cálculo (CU), que a su vez se dividen en uno o más elementos de procesamiento (PE). Los cálculos en un dispositivo de cómputo se producen dentro de los elementos de procesamiento, como se observa en la Figura 3.14.

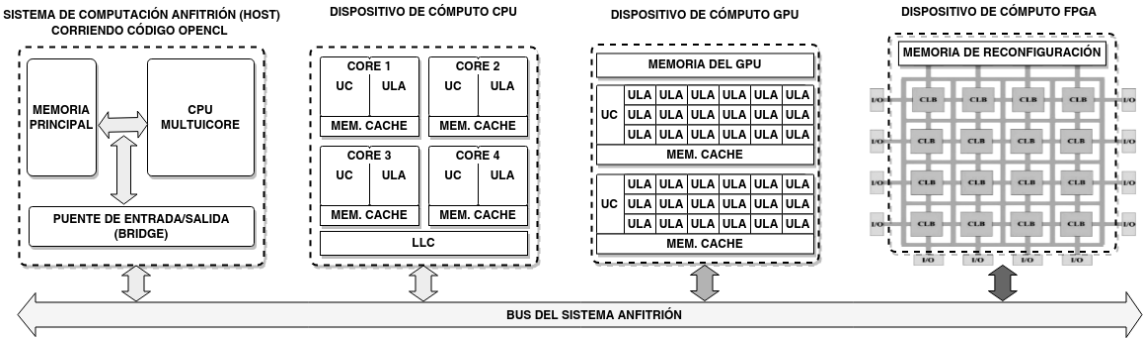


Figura 3.14: Configuración del Modelo de Plataforma de un Sistema de Computación Heterogénea corriendo código OpenCL. Fuente: Elaborado por el autor.

Una aplicación OpenCL se implementa como código en el Host o Anfitrión y como código del kernel en el dispositivo de cómputo OpenCL. La parte de código Host de una aplicación OpenCL se ejecuta en un

procesador Host según los modelos nativos de la plataforma Host. El código Host de la aplicación OpenCL envía el código kernel como comandos desde el Host hacia los dispositivos OpenCL. Un dispositivo OpenCL ejecuta el cálculo de los comandos en los elementos de procesamiento dentro del dispositivo. Es decir, en el anfitrión se ejecuta el código OpenCL que se ha escrito en C/C++ o cualquier lenguaje de programación soportado por el sistema.

Este mismo programa se ejecuta en el procesador del computador Anfitrión, siendo gestionado por el sistema operativo y ejecutado localmente, y coordinando las secciones de código OpenCL en los demás dispositivos que reciben datos y control a partir del código OpenCL.

Hay un compilador específico de OpenCL para el CPU, el GPU (con tarjetas aceleradoras NVidia, Intel o AMD, por mencionar algunas) o FPGA (con tarjetas aceleradoras de Xilinx, Altera, AMD, Intel, etc.).

Todos los tipos de dispositivos de cómputo OpenCL admiten el modelo de plataforma, de ejecución, de programación y de memoria de OpenCL y las API utilizadas en OpenCL para gestionar estos dispositivos.

3.5.2 *MODELO DE EJECUCIÓN (EXECUTION MODEL) DE OPENCL*

El modelo de ejecución de OpenCL [Khronos OpenCL Working Group, 2023] se define en términos de dos unidades de ejecución distintas: los kernels que se ejecutan en uno o más dispositivos OpenCL y un programa host que se ejecuta en el Host. En lo que respecta a OpenCL, los kernels son las secciones de código o tareas donde se produce el "trabajo" asociado a un cálculo. Este trabajo se produce a través de elementos de trabajo (work item) que se ejecutan en grupos (work group).

Un kernel se ejecuta dentro de un contexto bien definido gestionado por el Host. El contexto define el entorno en el que se ejecutan los kernels. Incluye los siguientes recursos:

- **Dispositivos (Devices):** Uno o más dispositivos expuestos por la plataforma OpenCL.
- **Objetos de Kernel (Kernel Objects):** Las funciones OpenCL con los valores de sus argumentos asociados que se ejecutan en los dispositivos OpenCL.
- **Objetos de Programa (Program Objects):** El código fuente del programa y el ejecutable que implementan los Kernels.
- **Objetos de Memoria (Memory Objects):** Las variables visibles para el Host y los dispositivos OpenCL. Las instancias de los Kernels operan sobre estos objetos mientras se ejecutan.

El programa Anfitrión o Host utiliza la API OpenCL para crear y gestionar el contexto. Las funciones de la API OpenCL permiten al Host interactuar con un dispositivo a través de una cola de comandos. Cada cola de comandos está asociada a un único dispositivo. Los comandos colocados en la cola de comandos se clasifican en uno de estos tres tipos:

- **Comandos de Encolado de Kernel (Kernel-enqueue commands:):** Pone en cola un kernel para su ejecución en un dispositivo.
- **Comandos de Memoria (Memory commands):** Transfieren datos entre el host y la memoria del dispositivo, entre objetos de memoria, o asignan y desasignan objetos de memoria. o asignar y desasignar objetos de memoria del espacio de direcciones del host.
- **Comandos de Sincronización (Synchronization commands):** Puntos de sincronización explícitos que definen restricciones de orden entre comandos.

Además de los comandos enviados desde la cola de comandos del Host, un Kernel que se ejecuta en un dispositivo puede enviar comandos a una cola de comandos del dispositivo. Esto da lugar a que los Kernels hijos en cola por un núcleo se ejecuten en un dispositivo (el núcleo padre). Independientemente de si la cola de comandos reside en el Host o en un dispositivo, cada comando pasa por seis estados: En cola (Queued), Enviado (Submitted), Listo (Ready), Ejecutandose (Running), Finalizado (Ended) y Completado (Complete) [Khronos OpenCL Working Group, 2023].

3.5.3 *MODELO DE MEMORIA (MEMORY MODEL) DE OPENCL*

El modelo de memoria de OpenCL [Khronos OpenCL Working Group, 2023] describe la estructura, el contenido y el comportamiento de la memoria expuesta por una plataforma OpenCL mientras se ejecuta un programa OpenCL. El modelo permite a un programador gestionar los valores de la memoria mientras se ejecutan el programa Host y varias instancias del Kernel en los dispositivos.

Un programa OpenCL define un contexto que incluye un Host, uno o más dispositivos, colas de comandos y la memoria expuesta dentro del contexto. El programa Host se ejecuta como uno o varios hilos Host gestionados por el sistema operativo que se ejecuta en el Host (cuyos detalles se definen fuera de OpenCL).

Puede haber varios dispositivos en un mismo contexto que tengan acceso a objetos de memoria definidos en OpenCL. En un mismo dispositivo, varios grupos de trabajo (work-group) pueden ejecutarse en paralelo con actualizaciones de memoria que podrían solaparse. Por último, dentro de un mismo grupo de trabajo, varios elementos de trabajo (work-item) se ejecutan simultáneamente, de nuevo con actualizaciones de memoria que pueden solaparse.

El modelo de memoria debe definir con precisión cómo interactúan los valores de la memoria vistos desde cada una de estas unidades de ejecución para que un programador pueda gestionar la correctitud de los programas OpenCL. Definimos el modelo de memoria en cuatro partes:

- **Regiones de memoria (Memory regions):** Las distintas memorias visibles para el Host y los dispositivos de cómputo que comparten un contexto.
- **Objetos de memoria (Memory objects):** Los objetos definidos por la API OpenCL y su gestión por el Host y los dispositivos.
- **Memoria virtual compartida (Shared Virtual Memory):** Un espacio virtual de direcciones expuesto tanto al Host como a los dispositivos dentro de un contexto. Nota: SVM no existe antes de la versión 2.0.
- **Modelo de coherencia (Consistency Model):** Reglas que definen qué valores se observan cuando múltiples unidades de ejecución cargan datos de la memoria, además de las operaciones atómicas que restringen el orden de las operaciones de memoria y definen las relaciones de sincronización.

La memoria en OpenCL se divide en dos partes:

- **Memoria del Host (Host Memory):** Es la memoria directamente disponible para el Host. El comportamiento detallado de la memoria se define fuera de OpenCL. Los objetos de memoria se mueven entre el Host y los dispositivos a través de funciones dentro de la API OpenCL o a través de una interfaz de memoria virtual compartida.
- **Memoria de dispositivo (Device Memory):** Es la memoria directamente disponible para los Kernels que se ejecutan en los dispositivos de cómputo OpenCL.

Además, la memoria de dispositivo consta de cuatro espacios de direcciones o regiones de memoria, como se muestra en la Figura 3.15:

- **Memoria global:** Esta región de memoria permite el acceso en modo lectura/escritura a todos los elementos de trabajo (work-item) en todos los grupos de trabajo (work-group) que se ejecutan en cualquier dispositivo dentro de un contexto. Los elementos de trabajo pueden leer o escribir en cualquier elemento de un objeto de memoria (memory object). Las lecturas y escrituras en la memoria global pueden almacenarse en memoria caché dependiendo de las capacidades del dispositivo.
- **Memoria constante:** Es una región de memoria global que permanece constante durante la ejecución de una instancia de Kernel (kernel-instance). El Host asigna e inicializa los objetos de memoria colocados en la memoria constante.
- **Memoria local:** Es una región de memoria local a un grupo de trabajo. Esta región de memoria puede utilizarse para asignar variables que son compartidas por todos los elementos de trabajo en ese grupo de trabajo.
- **Memoria privada:** Es una región de memoria privada de un elemento de trabajo. Las variables definidas en la memoria privada de un elemento de trabajo no son visibles para otro elemento de trabajo.

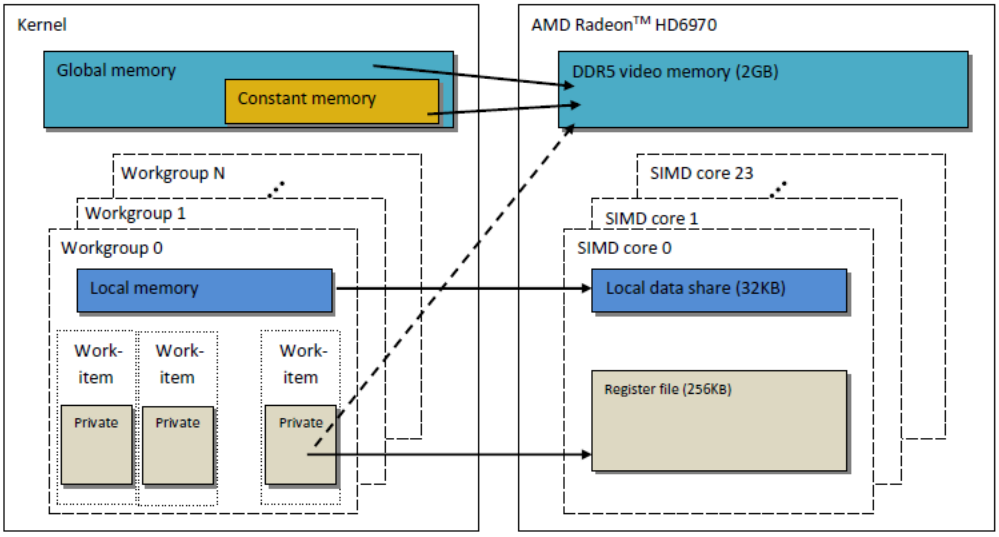


Figura 3.15: Correlación entre la estructura de memoria de una tarjeta GPU comercial y el modelo de memoria abstracto de OpenCL. Fuente: www.mql5.com

En resumen, las memorias locales y privadas siempre están asociadas a un dispositivo concreto. Las memorias globales y constantes sin embargo, se comparten entre todos los dispositivos dentro de un contexto determinado. Un dispositivo OpenCL puede incluir una caché para permitir un acceso eficiente a estas memorias compartidas.

3.5.4 *MODELO DE PROGRAMACIÓN (PROGRAMMING MODEL) DE OPENCL*

El framework de programación OpenCL [Khronos OpenCL Working Group, 2023] permite la programación de aplicaciones utilizando un Host y uno o más dispositivos de cómputo OpenCL como un único sistema heterogéneo de computación paralela. El framework contiene los siguientes componentes:

- **Capa de plataforma OpenCL (OpenCL Platform layer):** La capa de plataforma permite al programa anfitrión descubrir los dispositivos OpenCL y sus capacidades y crear contextos.
- **Tiempo de ejecución OpenCL (OpenCL Runtime):** El tiempo de ejecución permite al programa anfitrión manipular los contextos una vez creados.
- **Compilador OpenCL (OpenCL Compiler):** El compilador OpenCL crea ejecutables de programas que contienen Kernels OpenCL. El compilador OpenCL puede crear ejecutables de programas a partir de código fuente OpenCL C/C++, el lenguaje intermedio SPIR-V u objetos binarios de programa específicos del dispositivo, en función de las capacidades de un dispositivo. Algunas implementaciones pueden admitir otros lenguajes de Kernel o lenguajes intermedios.

La API de OpenCL tiene funciones para identificar los dispositivos que conforman el sistema heterogéneo, compilar programas, enviar y recibir datos y ejecutar estos programas en el dispositivo de cómputo elegido.

Es necesario tener previamente preparado el ambiente para editar, compilar y correr programas en OpenCL. Básicamente este proceso implica: a) Crear el código que se desea ejecutar utilizando un lenguaje de programación de alto nivel que soporte OpenCL. Generalmente este programa se crea el programa de Control del Anfitrión (Host) y el programa Kernel de los dispositivos, ambos en el Host; b) Crear los datos que se desean procesar en la arquitectura heterogénea; c) Inicializar los dispositivos de cómputo del sistema usando el código OpenCL o OCL; d) Utilizar la API de OpenCL para transferir datos y los comandos de control para la ejecución de la sección de código en los dispositivos de cómputo; y e) Recuperar todos los datos procesados por los dispositivos de cómputo.

En concreto, a continuación se resumen los pasos y secciones principales para codificar una aplicación OpenCL sencilla:

1. Descubrir las caracteristicas de la plataforma y dispositivos (Discovering the platform and devices).
2. Inicializar las variables de configuración de la plataforma (Set the platform configuration variables).
3. Obtener los datos y carga de trabajo a procesar (Get data and workload for processing).
4. Creación de un contexto de procesamiento (Creating a processing context).
5. Creación de una cola de comandos por dispositivo (Creating a command-queue per device).
6. Creación de objetos de memoria (buffers) para contener datos (Creating memory objects (buffers) to hold data).
7. Copia de los datos de entrada en el dispositivo (Copying the input data onto the device).
8. Creación y compilación de un programa a partir del código fuente OpenCL C/C++ (Creating and compiling a program from the OpenCL C/C++ source code).
9. Extracción del núcleo del programa (Extracting the kernel from the program).
10. Ejecución del núcleo (Executing the kernel).
11. Copia de los datos de salida al Host (Copying output data back to the Host).
12. Liberación de los recursos OpenCL (Releasing the OpenCL resources).

Un ejemplo de estructura genérica de un programa en OpenCL se puede ver en 3.7.

```
/*#############################################################################################
# Autor: Carlos Acosta (25/02/2022)
# Nombre del archivo: opencl-ejemplo1.cpp
# Función: Ejemplo en openCL/C++ que muestra la estructura básica de un programa para CPU y GPU.
#############################################################################################*/
#include <stdio.h>
#include <iostream>
#include <opencl.h>   // Librerías openCL

//Biblioteca OpenCL
using OpenCLTemplate; // Librerías openCL
 /* Global declaration */
 int *vectorIn, *vectorOut;
 int vector_size;

 // INICIO DEL PROGRAMA
 public static void Main(string[] args) {
  // Inicialización de los dispositivos CL disponibles (CPUs, GPUs, FPGAs, etc.)
  CLCalc.InitCL();
  /* Variables asociadas a los dispositivos */
  cl_uint num_devices_returned;
  cl_device_id devices[2];
  channel float cpu_gpu_channel;
  func_type *cpu_KernelTask, *gpu_KernelTask;
  // Creación de las variables que van a pasarse al dispositivo
  // Estas variables son memoria local, es decir, memoria del anfitrión
  float[] vectorIn  = new float[] { 1.0f, 2.0f, 3.0f, 4.0f };
  float[] vectorOut = new float[] {vector_size};
  /* Obtener apuntadores a los dispositivos */
  err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &devices[0], &num_devices_returned);
  err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &devices[1], &num_devices_returned);
  /* Se el contexto para los dispositivos de cómputo */
  cl_context context;
  context = clCreateContext(0, 2, devices, NULL, NULL, &err);
  /* Se crean las colas de comandos para los dispositivos */
  cl_command_queue queue_gpu, queue_cpu;
  device_gpu = clCreateCommandQueue(context, devices[0], 0, &err);
  device_cpu = clCreateCommandQueue(context, devices[1], 0, &err);
  /* Se definen los Kernels para los dispositivos */
  __kernel void cpu_KernelTask(__global int* &vectorIn, int vector_size) {
      // Código aquí
  }
  __kernel void gpu_KernelTask(__global int* &vectorIn, int vector_size) {
      // Código aquí
  }
  /* Se arranca la ejecución paralela de los Kernels (tasks) de los dispositivos CPU-GPU */
  clEnqueueTask(device_cpu, *cpu_KernelTask);
  clEnqueueTask(device_gpu, *gpu_KernelTask);
  /* Se finalizan los Kernels de los dispositivos */
  clFinish(device_gpu); // Kernel del GPU
  clFinish(device_cpu); // Kernel del CPU
}// FIN DEL PROGRAMA
```

Código fuente 3.7: Estructura genérica en OpenCL donde se muestra las secciones que conforman un programa para cómputo heterogéneo. Fuente: Elaborado por el autor.

3.5.5 *EJEMPLO PRÁCTICO DE PROGRAMACIÓN PARALELA HETEROGÉNEA USANDO OPENCL*

En esta sección se muestra el funcionamiento de OpenCL mediante un programa ejemplo. Para ello, se tiene una plataforma de computación Anfitrión con una configuración de un CPU Intel Core i7 de 4 núcleos físicos con 4 GB de memoria RAM y una tarjeta GPU NVidia Geforce GTX 580 de 512 núcleos con 3 GB de memoria de video. También, se usa la distribución del sistema operativo Linux Ubuntu LTS 20.04 y el editor de código fuente Visual Studio Code.

Además, como paso previo se muestra el proceso de preparación del ambiente para compilar y correr los ejemplos OpenCL mediante la instalación del compilador GNU C/C++, el depurador Git, el script CMake, la librería y los drivers del CPU y del GPU con soporte **OpenCL**.

```
$> sudo apt update; apt -y upgrade
$> sudo apt install build-essential -y
$> sudo apt install git -y
$> sudo apt install cmake -y
$> sudo apt install clinfo opencl-headers ocl-icd-opencl-dev -y
$> clinfo -l
$> sudo reboot
```

Otra alternativa para preparación del entorno de programación en OpenCL/C++ en Linux Ubuntu LTS 22.04 es el siguiente:

```
> mkdir neo
> cd neo
> wget https://github.com/intel/intel-graphics-compiler/releases/download/igc-1.0.12149.1 \
intel-igc-core_1.0.12149.1_amd64.deb
> wget https://github.com/intel/intel-graphics-compiler/releases/download/igc-1.0.12149.1 \
intel-igc-opencl_1.0.12149.1_amd64.deb
> wget https://github.com/intel/compute-runtime/releases/download/22.39.24347 \
intel-level-zero-gpu-dbgsym_1.3.24347_amd64.ddeb
> wget https://github.com/intel/compute-runtime/releases/download/22.39.24347 \
intel-level-zero-gpu_1.3.24347_amd64.deb
> wget https://github.com/intel/compute-runtime/releases/download/22.39.24347 \
intel-opencl-icd-dbgsym_22.39.24347_amd64.ddeb
> wget https://github.com/intel/compute-runtime/releases/download/22.39.24347 \
intel-opencl-icd_22.39.24347_amd64.deb
> wget https://github.com/intel/compute-runtime/releases/download/22.39.24347/libigdgmm12_22.2.0_amd64.deb
> sudo apt install ./*.deb
> /usr/bin/clinfo -l
> sudo reboot
```

En referencia a la instalación del entorno de programación OpneCL no hay que olvidar que es necesario instalar los drivers OpenCL del CPU, de la tarjeta GPU y de la tarjeta FPGA según la marca (plataforma) y modelo de cada una. A continuación se compila y corre un programa ejemplo en OpenCL (mostrado en el código fuente 3.8) que suma dos (2) vectores de números en punto flotante:

```
//
// Ejemplo OpenCL: Suma dos vectores A y B de números en punto flotante
// Fuente        : http://www.eriksmistad.no/getting-started-with-opencl-and-gpu-computing/
// openCL headers
#ifdef __APPLE__
#include <OpenCL/opencl.h>
#else
#include <CL/cl.h>
#endif
#include <stdio.h>
#include <stdlib.h>

#define MAX_SOURCE_SIZE (0x100000)
void printResult(float *vec_A, float *vec_B, float *vec_C, int tamVec);
void testResult(float *vec_A, float *vec_B, float *vec_C, int tamVec);

// Kernel definition
const char *kernelSource =
"__kernel void addVectors(__global const float *a,\n"
" __global const float *b,\n"
" __global float *c) { \n"
"   int gid = get_global_id(0);\n"
"   c[gid] = a[gid] + b[gid];\n"
" }\n";

int main(int argc, char ** argv) {
  // Variable declarations
    int SIZE = 1024;

  // Allocate memories for input arrays and output array.
  float *A = (float*)malloc(sizeof(float)*SIZE);  // Input
  float *B = (float*)malloc(sizeof(float)*SIZE);  // Input
  float *C = (float*)malloc(sizeof(float)*SIZE);  // Output
  // Initialize values for array members.
  int i = 0;
  for (i = 0; i < SIZE; ++i) {
    A[i] = i+1;
    B[i] = (i+1)*2;
  }
  // Load kernel from file vecAddKernel.cl
  FILE *kernelFile;
  char *kernelSource;
  size_t kernelSize;
  kernelFile = fopen("vecAddKernel.cl", "r");
  if (!kernelFile) {
    fprintf(stderr, "No file named vecAddKernel.cl was found\n");
    exit(-1);
  }
  kernelSource = (char*)malloc(MAX_SOURCE_SIZE);
  kernelSize = fread(kernelSource, 1, MAX_SOURCE_SIZE, kernelFile);
  fclose(kernelFile);
  // Getting platform and device information
  cl_platform_id platformId = NULL;
  cl_device_id deviceID = NULL;
  cl_uint retNumDevices;
  cl_uint retNumPlatforms;
  cl_int ret = clGetPlatformIDs(1, &platformId, &retNumPlatforms);
  ret = clGetDeviceIDs(platformId, CL_DEVICE_TYPE_DEFAULT, 1, &deviceID, &retNumDevices);
  // Creating context.
  cl_context context = clCreateContext(NULL, 1, &deviceID, NULL, NULL,  &ret);
  // Creating command queue
  cl_command_queue commandQueue = clCreateCommandQueue(context, deviceID, 0, &ret);
  // Memory buffers for each array
```

Código fuente 3.8: Ejemplo de código en OpenCL que suma dos vectores de números en punto flotante usando el GPU. Fuente: http://www.eriksmistad.no.

```
64   cl_mem aMemObj = clCreateBuffer(context, CL_MEM_READ_ONLY, SIZE * sizeof(float), NULL, &ret);
65   cl_mem bMemObj = clCreateBuffer(context, CL_MEM_READ_ONLY, SIZE * sizeof(float), NULL, &ret);
66   cl_mem cMemObj = clCreateBuffer(context, CL_MEM_WRITE_ONLY, SIZE * sizeof(float), NULL, &ret);
67   // Copy lists to memory buffers
68   ret = clEnqueueWriteBuffer(commandQueue, aMemObj, CL_TRUE, 0, SIZE * sizeof(float), A, 0, NULL, NULL);;
69   ret = clEnqueueWriteBuffer(commandQueue, bMemObj, CL_TRUE, 0, SIZE * sizeof(float), B, 0, NULL, NULL);
70   // Create program from kernel source
71   cl_program program = clCreateProgramWithSource(context, 1, (const char **)&kernelSource,
72                                                  (const size_t *)&kernelSize, &ret);
73   // Build program
74   ret = clBuildProgram(program, 1, &deviceID, NULL, NULL, NULL);
75   // Create kernel
76   cl_kernel kernel = clCreateKernel(program, "addVectors", &ret);
77   // Set arguments for kernel
78   ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&aMemObj);
79   ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&bMemObj);
80   ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&cMemObj);
81   // Execute the kernel
82     printf("Processing ...! \n");
83   size_t globalItemSize = SIZE;
84   size_t localItemSize = 64; // globalItemSize has to be a multiple of localItemSize. 1024/64 = 16
85   ret = clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL, &globalItemSize,
86                                &localItemSize, 0, NULL, NULL);
87   // Read from device back to host.
88   ret = clEnqueueReadBuffer(commandQueue, cMemObj, CL_TRUE, 0, SIZE * sizeof(float), C, 0, NULL, NULL);
89
90   // Write result
91   for (i = 0; i < SIZE; ++i) {
92     printf("%f + %f = %f\n", A[i], B[i], C[i]);
93   }
94   // Test if correct answer
95   for (i = 0; i < SIZE; ++i) {
96     if (C[i] != (A[i] + B[i])) {
97       printf("Something didn't work correctly! Failed test. \n");
98       break;
99     }
100  }
101  if (i == SIZE) {
102    printf("Everything seems to work fine! \n");
103  }
104  // Clean up, release memory.
105  ret = clFlush(commandQueue);
106  ret = clFinish(commandQueue);
107  ret = clReleaseCommandQueue(commandQueue);
108  ret = clReleaseKernel(kernel);
109  ret = clReleaseProgram(program);
110  ret = clReleaseMemObject(aMemObj);
111  ret = clReleaseMemObject(bMemObj);
112  ret = clReleaseMemObject(cMemObj);
113  ret = clReleaseContext(context);
114  free(A);
115  free(B);
116  free(C);
117
118  printf("... Releasing and ending program!\n");
119 return 0;
120 }
```

(Cont. Codigo fuente 3.8) Ejemplo de código en OpenCL que suma dos vectores de números en punto flotante usando el GPU. Fuente: http://www.eriksmistad.no.

Al compilar y correr el programa ejemplo en OpenCL mostrado en el código fuente 3.8 obtenemos la siguiente salida:

```
$> g++ vecAdd-2 (ejemplo opencl).cpp -w -o vecAdd-2 (ejemplo opencl).out -lOpenCL
$> ./vecAdd-2 (ejemplo opencl).out
    Processing ...!
    Everything seems to work fine!
    ... Releasing and ending program!
$>
```

OpenCL está diseñado para ser portable a otras arquitecturas y diseños de hardware. OpenCL ha utilizado en su núcleo un lenguaje de programación basado en C99 y sigue reglas basadas en esa herencia. La aritmética de coma flotante se basa en los estándares IEEE-754 e IEEE-754-2008. Los objetos de memoria, los punteros y la memoria débilmente ordenada se han diseñado para ofrecer la máxima compatibilidad con las arquitecturas de memoria discreta implementadas por los dispositivos OpenCL. Las colas de comandos y las barreras permiten la sincronización entre el Host y los dispositivos OpenCL. El diseño, las capacidades y las limitaciones de OpenCL son en gran medida un reflejo de las capacidades del hardware subyacente [Kaeli et al., 2015].

En conclusión, se ha seleccionado el lenguaje OpenCL por ser un lenguaje de programación de alto nivel orientado a la programación paralela de sistemas de arquitectura heterogénea. El propósito es aprovechar su síntaxis, semántica y la experiencia que el programador posee de un lenguaje compatible con C/C++. Esto reduce la curva de aprendizaje de la herramienta y ayuda a utilizar los recursos del framework con mayor eficiencia y productividad. Además, provee el soporte necesario para construir una herramienta de programación que haga transparente el codiseño y programación de aplicaciones de alto rendimiento que involucran componentes de software en CPU-GPU y de hardware en FPGA.

3.6 CONSIDERACIONES FINALES DEL CAPÍTULO

La revisión de trabajos previos en el capítulo anterior y de los conceptos involucrados en la solución propuesta muestran que es factible emplear el enfoque de esqueletos algorítmicos para agregar el nivel de abstracción necesario que permita encapsular y ocultar los detalles asociados al paralelismo, el diseño y reconfiguracion del FPGA.

Este enfoque permite que la heterogeneidad del proceso de codiseño de una aplicación se aborde ahora como un proceso homogéneo y transparente. Esto se lograría permitinedo el desarrollo de los componentes de hardware y software del sistema al mismo nivel de abstracción, además del movimiento transparente de funcionalidades entre el CPU-GPU y el FPGA. También, habilita la interacción de estos componentes usando interfaces de comunicación, independientemente de sus implementaciónes.

Es así que, el enfoque basado en esqueletos algorítmicos permite encapsular y reusar tareas de procesamiento en forma de componentes de hardware y software escondiendo los detalles de bajo nivel asociados al paralelismo y la reconfiguración. Así, el software y el hardware se pueden codificar de manera que sean independientes de una arquitectura reconfigurable particular y en consecuencia se puede lograr portabilidad con buen rendimiento.

Sin embargo, esto implica apoyarse en una herramienta, por ejemplo como un lenguaje de programación de alto nivel, que haga posible construir una capa de abstración que permita la descripción, diseño e implementación homogénea de los componentes paralelos de hardware y software que conforman la aplicación embebida.

Por tanto, para integrar los esqueletos algoritmicos dentro de la metodologia de codiseño es necesario emplear un lenguaje que provea el suficiente nivel de abstracción que permita la descripción o especificación de los componentes de hardware y software, sin favorecer una implementación particular.

Así que, es de interés para el presente trabajo de tesis usar los esqueletos algorítmicos en forma de funciones de orden superior para apoyar el codiseño, es decir, la programación homogénea de tareas en hardware y software de la aplicación. Además, esto permite la exploración de las diferentes configuraciones o espacios de diseño de la aplicación mediante el intercambio transparente de funcionalidades entre componentes de hardware y software.

Ahora, teniendo en cuenta que los lenguajes existentes apuntan a describir software (lenguajes de programación) o a describir hardware (lenguajes de descripción de hardware), y aunque ambos tipos de lenguajes apuntan a describir "procesos de computación", tienen como base modelos de programación distintos. Esta es la razón por la cual el mecanismo de encapsulamiento es tan importante para la selección del lenguaje de programación, dado que será el elemento clave para lograr construir la capa de abstracción que ayudará en el codiseño homogéneo y transparente de aplicaciones embebidas sobre plataformas reconfigurables del tipo CPU-GPU-FPGA.

Entonces, es evidente que la implementación de las propiedades de las funciones de orden superior son un recurso deseable en el lenguaje de programación de alto nivel seleccionado. Esto, porque es la forma de proporcionar esqueletos algorítmicos como un mecanismo de encapsulación que permite una expresión elegante, estructurada e implícita del paralelismo y la reconfiguración con un alto nivel de abstracción.

Esta es la razón por la cual el mecanismo de encapsulamiento es tan importante en un lenguaje de programación dado que es la manera de construir una capa de abstracción de alto nivel para el codiseño de aplicaciones.

Capítulo 4

SkeletonCoRe: Interfaz de Codiseño y Programación de Aplicaciones Paralelas Heterogéneas basada en Esqueletos Algorítmicos Reconfigurables

"El logro más impresionante de la industria del software es su continua anulación de los constantes y asombrosos logros de la industria del hardware."

Henry Petroski

4.1 INTRODUCCIÓN

Habiendo revisado la literatura de trabajos relacionados al tema de esta investigación, así como los fundamentos teóricos y tecnológicos asociados a la computación paralela, al paralelismo estructurado con esqueletos algorítmicos, al co-diseño de aplicaciones hardware/software, a los FPGA y al lenguaje de programación paralela heterogénea OpenCL; ahora se presenta y explica el diseño e implementación de la herramienta que integra todos estos elementos en una solución denominada SkeletonCoRe [Acosta-León and Suros, 2022]. La metodología de desarrollo empleada para la construcción de la solución propuesta es la siguiente:

I. Fase de Análisis:

1. **Determinación de los criterios de análisis, diseño e implementación** de la PAPI *SkeletonCoRe* usando el enfoque orientado a objetos y la teoría de funciones de orden superior.

II. Fase de Diseño:

1. **Definición de la arquitectura y flujo de diseño** de aplicaciones paralelas que conforma la PAPI *SkeletonCoRe*.

2. **Diseño y descripción de los módulos y algoritmos** que conforman los bloques de construcción de los Esqueletos Reconfigurables y demás componentes de la PAPI *SkeletonCoRe*. Para ello, se aplica como técnica el enfoque orientado a objetos y los algoritmos se describen usando un dialecto pseudoformal similar al usado por el lenguaje C/C++.

III. Fase de Implementación:

1. **Descripción de los recursos de hardware y software**, es decir, los dispositivos del sistema de computación heterogénea compatibles con OpenCL con sus drivers y el API OpenCL y el lenguaje C/C++ como soporte de programación. Todos estos utilizados para la implementación de los Esqueletos Reconfigurables.
2. **Instalación, configuración y prueba** de las herramientas de software y hardware que se utilizan en la programación y corrida de la aplicación implementada con esqueletos reconfigurables en la plataforma heterogénea.
3. **Implementación de los códigos fuentes de los Esqueletos Reconfigurables** y demás componentes de apoyo de *SkeletonCoRe* usando programación orientada a objetos con el lenguaje C++ y el API OpenCL.

IV. Fase de Prueba y evaluación:

1. **Diseño de los experimentos** y de las configuraciones heterogéneas de computación usadas para la evaluación de la abstracción, funcionalidad y rendimiento de los Esqueletos Reconfigurables.
2. **Descripción de la aplicación de prueba** usada para la evaluación de los Esqueletos Reconfigurables de *SkeletonCoRe*.
3. **Ejecución de los experimentos y evaluación**, tabulación y graficación de datos, interpretación y discusión de los resultados.

La herramienta de codiseño y programación de alto nivel propuesta en esta investigación proporciona un catálogo de esqueletos algorítmicos reconfigurables. Se presenta como una interfaz de programación de aplicaciones paralelas embebidas en sistemas de computación heterogénea reconfigurables (Parallel Applications Programming Interface for Reconfiguable Computing, PAPI), lo cual constituye el objeto de la presente investigación.

SkeletonCoRe (como se muestra en la Figura 4.4) es una arquitectura jerárquica compuesta por una pila (stack) de capas de abstracción de alto nivel que encapsulan desde objetos o componentes de bajo nivel hasta patrones de alto nivel que estructuran diferentes modelos o algorítmos de computación paralela que pueden ser reconfigurables en FPGA. Estos patrones de cómputo paralelo de la capa de mas alto nivel se diseñan usando el enfoque de Esqueletos Algoritmicos de Cole.

Los esqueletos reconfigurables que conforman la Biblioteca o Librería de la interfaz de programación de aplicaciones paralelas o PAPI **SkeletonCoRe** se diseñan e implementan como funciones de orden superior los cuales se instancian secuencialmente dentro del programa paralelo que construye el programador en forma de plantillas que encapsulan el código paralelo usando el API openCL con el lenguaje C/C++ como soporte, donde cada plantilla toma datos y funciones como parámetros y retorna resultados. Estos parámetros definen internamente el comportamiento paralelo de cada esqueleto, donde se coordina de forma transparente la interacción entre las tareas de software en CPU-GPU y las tareas de hardware en FPGA.

Con base en lo argumentado en el capítulo anterior, se ha seleccionado el API OpenCL C/C++ por ser un lenguaje de programación de alto nivel orientado a la programación paralela de sistemas de arquitectura heterogénea.

El propósito de dicha elección es aprovechar su síntaxis, semántica y la experiencia que el programador común posee de un lenguaje compatible con C/C++. Esto reduce la curva de aprendizaje de la herramienta y ayuda a utilizar los recursos de la PAPI con mayor eficiencia y productividad. Además, OpenCl agrega recursos e instrucciones que explotan paralelismo que permiten diseñar y programar los componentes o tareas de software (en CPU-GPU) y de hardware (en FPGA) de la aplicación paralela de manera transparente y al mismo nivel de abstracción.

En definitiva, el enfoque de esqueletos algorítmicos usado en SkeletonCoRe permite encapsular y reusar tareas de procesamiento paralelo y esconder los detalles de bajo nivel asociados al paralelismo, el diseño digital y la reconfiguración. Así, los componentes, tareas o secciones de código de software y hardware de la aplicación paralela se pueden co-diseñar de manera que sean independientes de una arquitectura reconfigurable particular y en consecuencia se puede lograr portabilidad con buen rendimiento. Además, de esta manera se provee al programador el suficiente nivel de abstracción para mover fácilmente funcionalidades entre software y hardware durante la etapa de exploración de espacios de diseño de la aplicación.

4.2 DETERMINACIÓN DE CRITERIOS DE DESARROLLO DE LA PAPI SKELETONCORE

Aunque existen proyectos relacionados, el diseño e implementación de software sobre sistemas de computación reconfigurables tiende a ser un proceso heterogéneo y separado, desaprovechando así su mayor ventaja de flexibilidad y reconfigurabilidad.

Por eso, la meta del presente proyecto de tesis doctoral es, en términos generales, crear una herramienta que provea codiseño homogéneo de aplicaciones paralelas embebidas de alto rendimiento con transparencia, abstracción y escalabilidad, y así poder explorar diferentes escenarios de diseño para obtener la solución con el mejor rendimiento y abstracción posible.

4.2.1 *CRITERIOS DE ANÁLISIS DE LA INTERFAZ DE PROGRAMACIÓN SKELETONCORE*

La herramienta propuesta aplica los siguientes criterios de análisis, diseño y programación:

- **Complejidad:** Reduce la dificultad asociada a la programación paralela y diseño sobre el hardware del FPGA y facilita el acoplamiento del microprocesador (CPU-GPU) con el FPGA para su ejecución cooperativa y coordinada.
- **Transparencia:** Esconde al usuario los detalles y complejidades estructurales y funcionales del sistema que da solución al problema.
- **Abstracción:** Permite, mediante encapsulamiento, el codiseño homogéneo de los componentes de hardware y software al mismo nivel de abstracción.
- **Optimización:** Permite explorar espacios de diseño mediante el intercambio de funcionalidades entre los componentes de hardware y software de forma transparente, para elegir la configuración de la solución con mejor rendimiento.
- **Escalabilidad:** Explota el paralelismo a diferentes niveles de granularidad, en particular estructurandolo en torno a modelos algorítmicos de alto nivel como los esqueletos.

- **Rendimiento:** Acelera la ejecución de la aplicación mediante la implementación en el hardware del FPGA de las tareas (núcleos de cómputo o kernels) de la aplicación que realizan cómputo intensivo, separando las secciones de entrada/salida intensiva de datos.

4.2.2 *CRITERIOS DE ELECCIÓN DE OPENCL/C++ PARA DESARROLLAR SKELETONCORE*

Como se ha comentado anteriormente, para implementar los esqueletos algorítmicos reconfigurables como una PAPI (Parallel Applications Programming Interface) en el proceso de codiseño y construcción de aplicaciones paralelas en arquitecturas de computación heterogénea es necesario emplear una herramienta o un lenguaje que provea el suficiente nivel de abstracción que permita la descripción o especificación de los componentes de hardware y de software, sin favorecer una implementación particular.

Sin embargo, una dificultad actual es que la mayoría de los lenguajes existentes se orientan a describir software (HLL, lenguajes de programación de alto nivel) o a describir hardware (HDL, lenguajes de descripción de hardware). Aunque ambos tipos de lenguajes se usan para describir algoritmos o procesos de computación, tienen como base distintos modelos de programación.

Esta es la razón por la cual el mecanismo de encapsulamiento es tan importante en un lenguaje de programación dado que permite construir una capa de abstracción de alto nivel para el codiseño y programación de aplicaciones.

En este sentido, OpenCL está diseñado para ser portable a otras arquitecturas y diseños de hardware. OpenCL ha utilizado en su núcleo un lenguaje de programación basado en C99 y sigue reglas basadas en esa herencia. Asímismo, los objetos de memoria, los punteros y la memoria débilmente ordenada se han diseñado para ofrecer la máxima compatibilidad con las arquitecturas de memoria local implementadas por los dispositivos OpenCL.

La selección del API OpenCL basado en C/C++ se debe a que es un lenguaje de programación de alto nivel orientado a la programación paralela de sistemas de arquitectura heterogénea. El lenguaje C/C++ es de uso común, por ello se puede aprovechar su síntaxis, semántica y la experiencia que el programador posee de un lenguaje compatible. Esto reduce la curva de aprendizaje de la herramienta y ayuda a utilizar los recursos del framework con mayor eficiencia y productividad.

En consecuencia, OpenCL C/C++ provee un enfoque orientado a objetos y el soporte necesario para construir la herramienta de desarrollo de aplicaciones paralelas que se propone en este trabajo para hacer transparente el codiseño y programación de aplicaciones de alto rendimiento que involucran componentes de software en CPU-GPU y de hardware en FPGA.

Con base en lo anterior, en el presente trabajo se ha elegido el API de OpenCL del lenguaje de programacion C/C++ por cuanto hereda un conjunto de elementos y recursos de abstracción de la programación orientada a objetos e instrucciones de paralelismo explícito que permiten encapsular patrones paralelos dentro de esqueletos algoritmicos e implementarlos como funciones de orden superior. Dentro de estos elementos y recursos tenemos:

- **Reutilización de Código:** Cuando se diseñan correctamente las clases, se pueden usar en distintas partes del programa en diferentes proyectos. La propiedad de heredar las caracteristicas de un objeto mas general (métodos y atributos) permite ahorrar tiempo porque al crear una clase genérica las subclases que se definen a partir de ella heredarán las propiedades de la misma, de manera que no es necesario escribir esas funciones de nuevo. Además, al aplicar un cambio en la clase, todas las subclases lo adoptarán automáticamente.

- **Mayor modificabilidad:** Otra de las ventajas de la programación orientada a objetos es que permite añadir, modificar o eliminar nuevos objetos o funciones fácilmente para actualizar los programas, lo cual implica un ahorro de tiempo y esfuerzo para los programadores.
- **Facilidad de Detección de Errores en el Código:** En los lenguajes de programación orientada a objetos no es necesario revisar línea por línea el código para detectar un error. Gracias a la encapsulación los objetos son autónomos, de manera que es más fácil buscar el error cuando algo no funciona bien. Es decir, se eliminan los efectos colaterales debido a errores entre objetos, dado que éstos se limitan a los objetos donde ocurren.
- **Modularidad y Abstracción:** Una de las características de la programación orientada a objetos más interesantes es la modularidad ya que así un equipo de programación puede trabajar en múltiples objetos a la vez mientras se minimizan las posibilidades de que un programador duplique la funcionalidad de otro. Además, la abstracción se provee al poderse encapsular los detalles de implementacion a lo interno eel objeto. El trabajo modular también permite dividir los problemas en partes más pequeñas y simples que se pueden probar de manera independiente.
- **Flexibilidad debido al Polimorfismo:** El polimorfismo de la programación orientada a objetos permite que una sola función pueda cambiar de forma para adaptarse a cualquier clase donde se encuentre. De esta forma se ahorra tiempo de programación y se gana en versatilidad.

Sin embargo, la carateristica mas importante de OpenCL es que su **modelo de memoria, su modelo de ejecución y su modelo de computación**, permiten al programador manejar los dispositivos de procesamiento de la misma forma, a través de sus **drivers o interfaces**, es decir, los trata como arquitecturas homogéneas con la misma estructura y comportamiento, de ahi que los dispositivos de procesamiento que fabrican empresas como Intel, Apple, Altera, etc., deben fabricarlos usando el estandard que los hace compatibles con OpenCL (denominados dispositivos OpenCL). Ver Figura 4.1a y 4.1b.

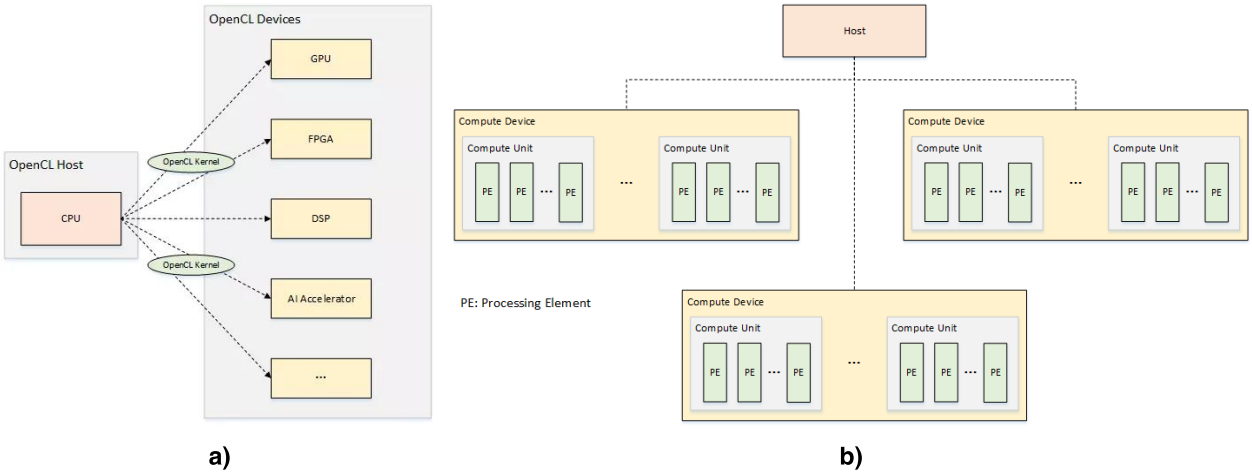


Figura 4.1: Proceso de funcionamiento de OpenCL basado en Modelos: Modelo de Arquitectura de la Plataforma. Fuente: Elaborado por el autor.

Asímismo, el modelo de ejecución de OpenCL permite que el programador organice la ejecución en función de un **programa Host** y varias **instancias del Kernel** en los dispositivos. Es así que, OpenCL define un

contexto que incluye un Host, uno o más dispositivos, colas de comandos y la memoria expuesta dentro del contexto (Ver Figura 4.2a y 4.2b).

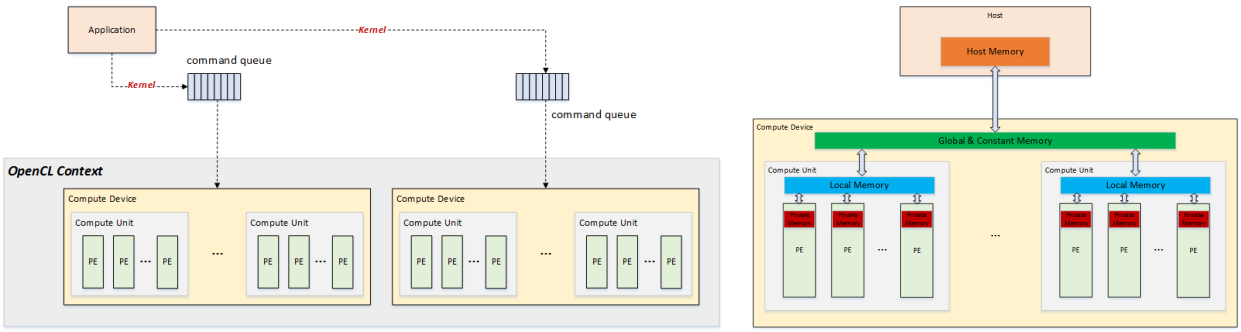


Figura 4.2: Proceso de funcionamiento de OpenCL basado en Modelos: Modelo de Memoria de los Dispositivos. Fuente: Elaborado por el autor.

En cuanto al **modelo de computación**, las unidades de ejecución implicadas en un programa Host se ejecutan como uno o varios hilos Host gestionados por el sistema operativo que se ejecuta en el Host (cuyos detalles se definen fuera de OpenCL). Además, en un mismo dispositivo, existen varios **grupos de trabajo (work-group)** que pueden ejecutarse en paralelo con actualizaciones de memoria que pueden solaparse. Por último, dentro de un mismo grupo de trabajo, varios **elementos de trabajo (work-item)** se ejecutan simultáneamente, de nuevo con actualizaciones de memoria que pueden solaparse (Ver Figura 4.3a y 4.3b).

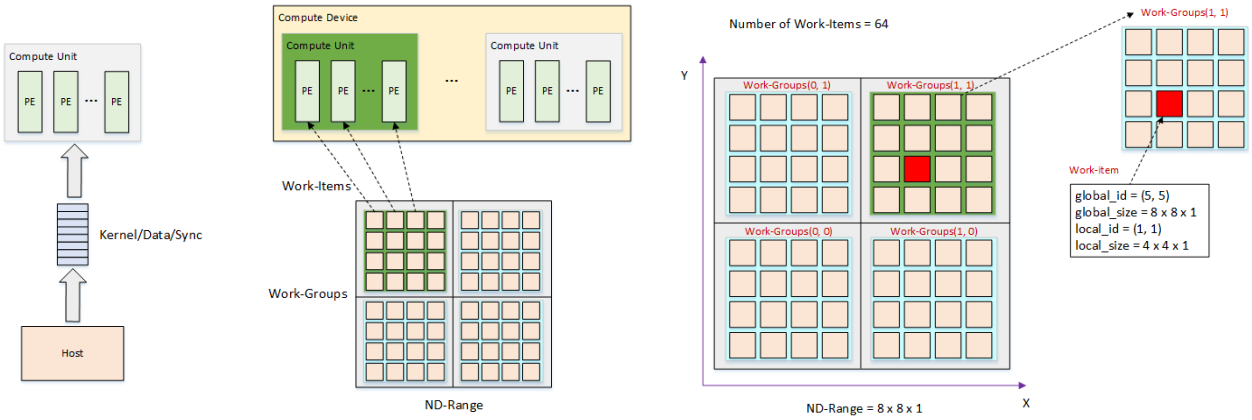


Figura 4.3: Proceso de funcionamiento de OpenCL basado en Modelos: Modelo de Computación de los Dispositivos. Fuente: Elaborado por el autor.

En consecuencia OpenCL se diseñó para ser portable a otras arquitecturas y diseños de hardware, siempre que sean compatibles con las especificaciones de OpenCL.

4.3 Definición de la arquitectura y flujo de diseño de SkeletonCore

4.3.1 *Descripción de la arquitectura de la API SkeletonCore*

La arquitectura de SkeletonCoRe y la metodología de Codiseño asociada se muestran en la Fig. 4.4a), comprende las siguientes capas:

- **Nivel 1: Capa Física de la Architectura Heterogénea.** Conformada por los dispositivos de computación de la arquitectura heterogénea: dispositivos microprogramables que corren algoritmos en software, como los CPUs y GPUs; y dispositivos reconfigurables que funcionan digitalmente como algoritmos en hardware, como los FPGAs y ASICs.

- **Nivel 2: Capa de Compilación/Síntesis e Interfase de Comunicación.** Provee las herramientas de compilación para producir código ejecutable en software, de síntesis para generar el archivo de configuración en hardware del FPGA y las respectivas interfaces hardware/software que controlan el intercambio de datos y control entre las tareas en hardware (componentes funcionando electrónicamente en FPGA) y las tareas en software (componentes corriendo en CPU y/o GPU) del sistema heterogéneo.

- **Nivel 3: Capa de Lenguaje de Programación de Alto Nivel.** Esta capa proporciona el soporte y los recursos a las capas superiores para construir las abstracciones de alto nivel. Esta capa es necesaria para el diseño e implementación de los patrones de cómputo paralelo como esqueletos algorítmicos, las interfaces de comunicación y los componentes o tareas en hardware y en software. Es decir, esta capa oculta los detalles de diseño del FPGA asociados a la síntesis, particionamiento, enrutamiento y colocación del componente de hardware y los detalles de software asociados al paralelismo explícito, es decir, oculta los detalles de implementación de las capas inferiores.

- **Nivel 4: Capa de Componentes de Hardware y Software.** Comprende los componentes de software (procesos e hilos) y hardware (procesadores y bloques digitales) que expresan explícitamente paralelismo. Los componentes de esta capa son usados por los esqueletos para interactuar de forma trasparente con el sistema de computación heterogénea (CPU-GPU-FPGA) para coordinar la comunicación, sincronización e intercambio de datos y, administración del paralelismo y de la reconfiguración.

- **Nivel 5: Capa de Esqueletos Algorítmicos Reconfigurables.** Es la capa que encapsula los componentes de harware y software como esqueletos algorítmicos, y esconde la implementacion de los detalles de bajo nivel de la Plataforma Reconfigurable (diseño lógico del FPGA, control de la reconfiguración, comunicación entre componentes de hardware y software, y paralelismo).

- **Nivel 6: Capa de Codiseño y Programación Hardware/Software de Aplicación del Usuario.** Es el nivel más alto de la PAPI SkeletonCoRe donde el programador diseña y codifica la aplicacion paralela de forma implícita usando secuencialmente las plantillas de esqueletos reconfigurables parametrizables que provee SkeletonCoRe.

Como complemento a la arquitectura antes descrita, en la Fig. 4.4b) se puede observar un diagrama que refleja las actividades asociadas en el proceso de Codiseño y Programación usuando SkeletonCoRe.

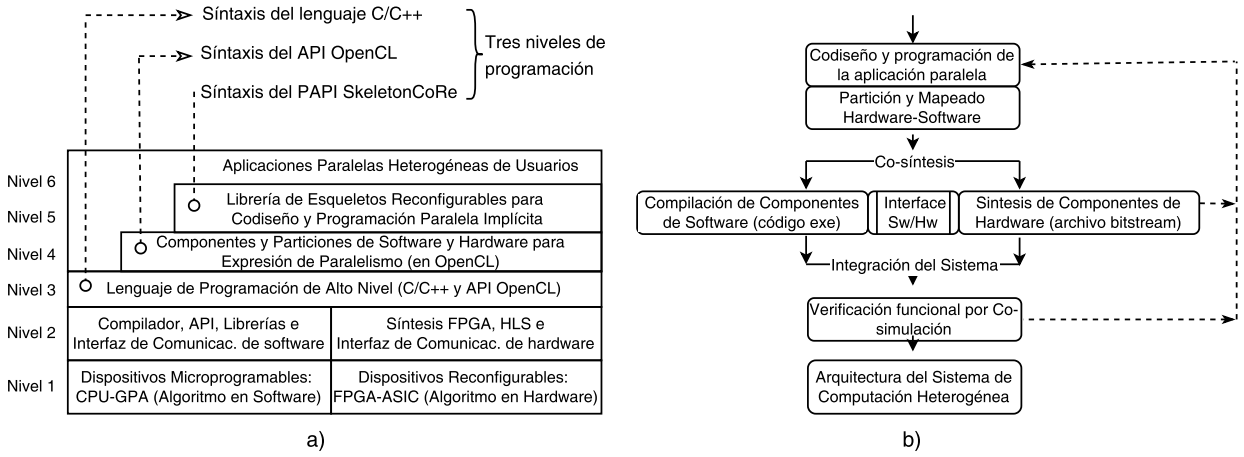


Figura 4.4: a) Arquitectura del PAPI SkeletonCoRe, b) Diagrama de la Metodología de Co-diseño y Programación de SkeletonCoRe. Fuente: Elaborado por el autor

4.3.2 *DESCRIPCIÓN DEL FLUJO DE DISEÑO CON LA API SKELETONCORE*

En la Figura 4.5 se ilustra con detalle el flujo de codiseño e implementación de una aplicación paralela usando los esqueletos de SkeletonCoRe. Aquí el programador usa los esqueletos (1) y proporcionar los parámetros apropiados para personalizar el esqueleto a una aplicación paralela específica. La biblioteca de esqueletos (2) es el repositorio con las implementaciones de los esqueletos algorítmicos de hardware y software. Se ha usado el API de openCL y de base un lenguaje de alto nivel como C/C++, para la programación de aplicaciones sobre el CPU-GPU y el FPGA de la plataforma heterogénea. En (3) y (4) el usuario puede toman decisiones para separar los componentes o tareas de software y hardware que explotan paralelismo/reconfiguración, y en (5) se aborda lo concerniente a la asignacion de un componente o tarea sobre el FPGA (harware) o sobre el CPU-GPU (software). Por último, en (6) se pueden probar previamente las particiones de hardware y software mediante co-simulación para verificar el funcionamiento y rendimiento de la aplicación paralela sobre la plataforma de computacion heterogenea, para finalmente integrar la aplicación.

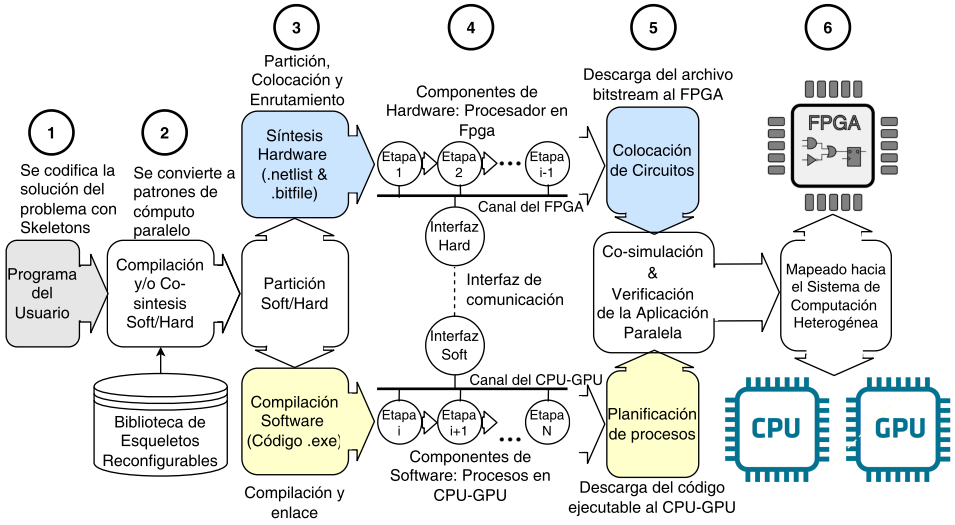


Figura 4.5: Flujo de diseño para crear una aplicación paralela usando la PAPI SkeletonCoRe. Fuente: Elaborado por el autor.

4.4 DISEÑO Y DESCRIPCIÓN DE LOS OBJETOS Y ALGORITMOS DE SKELETONCORE

En la presente sección se muestran los aspectos de diseño que soportan la construcción de los esqueletos y cómo se obtiene la abstracción y transparencia suficiente para ocultar los detalles del paralelsimo y diseño de FPGA que se explotan de forma implícita.

4.4.1 *DISEÑO DE LOS OBJETOS QUE ESTRUCTURAN LOS ESQUELETOS RECONFIGURABLES*

Es importante señalar que la programación basada en objetos implica un cambio de paradigma en la manera como se modela e implementa la solución de un problema.

En este sentido, y en el contexto del presente trabajo de investigación, la programación basada en objetos se ha usado para ocultar la estructura y funcionamiento de los esqueletos reconfigurables y demás componentes del **PAPI SkeletonCore**, y así lograr un alto nivel de abstracción y encapsulación. Es por ello que el diseño e implementación de los elementos que se organizan y estructuran en **SkeletonCore** se modelan en clases que, posteriormente, se concretan en objetos.

Mientras que la clase es la plantilla genérica que modela las categorias (datos y comportamiento) de un esqueleto algorítmico; el objeto es el modelo concreto de esqueleto que se crea al instanciar la clase. Esto agrega la capacidad de abstracción a la solución del problema.

Una ventaja evidente es la reutilización del código, lo que evita su duplicación. En este aspecto el principio de herencia hace que exista una jerarquización entre las clases de esqueletos que se definen y se reduzca las veces que se tienen que reescribir. Además, el principio de herencia permite ahorrar tiempo porque facilita la correspondencia entre clases y objetos.

Otra ventaja es el mecanismo de encapsulación de los datos y comportamiento del esqueleto, el cual hace posible que toda la información interna (privada) de un esqueleto quede dentro del mismo. En consecuencia, el acceso desde fuera se hace prácticamente imposible, logrando seguridad y transparencia.

En definitiva, este sistema permite que se controle qué datos y comportamiento son privados y cuáles son públicos. Esto permite un mayor control sobre el proceso de programación paralela reconfigurable.

También, es necesario mencionar el polimorfismo como mecanismo que permite diseñar los objetos esqueletos de tal forma que compartan datos y comportamientos con otros. El efecto que se consigue es que se pueden utilizar los objetos esqueletos algorítmicos de distintas maneras, básicamente, usando la sobrecarga del esqueleto. El poliformismo es una forma versátil de programar y adaptar los esqueletos algorítmicos a la aplicación paralela heterogénea.

A continuación se presenta el diseño algorítmico aproximado (descritos en pseudo C/C++) de las clases, objetos y métodos (datos y, funciones u operaciones) asociados a los elementos más importantes que organizan la estructura y funcionalidad general de la Interfaz para Programación de Aplicaciones Paralelas **(Parallel Application Programming Interface, PAPI) SkeletonCoRe** basada en esqueletos algorítmicos reconfigurables.

Estas clases, objetos y funciones que se describirán a continuación comprenden los modelos de diseño de: **a) "SkeletonCore Enviorenment"** para preparar y decubrir la Plataforma Heterogénea, es decir, los dispositivos de cómputo cpu, gpu y/o fpga existentes en la plataforma, **b) "Data Set"** para tener contenedores de diferentes tipos de datos a procesar (imágenes, matrices, vectores, etc.), **c) "Device Partition"** para la partición de cómputo con toda la información necesaria (datos y tareas) para que los dispositivos cpu, gpu y/o fpga realicen el procesamiento, **d) "Parallel Skeleton Frame"** para el modelo

genérico del esqueleto algoritmico pipeline, master/slave, etc., y **e) "Main Parallel Program"** para mostrar el uso genérico de los componentes y esqueletos reconfigurables de la PAPI SkeletonCoRe.

Es importante mencionar que el propósito de los modelos genéricos de la herramieta mostrados acá algoritmicamente es presentar los aspectos conceptuales más importantes de la PAPI SkeletonCoRe, e intencionalmente se dejan vacios los contenidos de algunos datos y operaciones que se definen al momento de su implementación (sólo declarados, sin definir). (Ver Figuras 4.6, 4.7, 4.8, 4.9 y 4.10).

A) Modelo de Código de "Enviorenment Skeleton": Este modelo considera la operación para obtener la información de los dispositivos de cómputo existentes en la Plataforma Heterogénea de Computación Reconfigurable, acción previa necesaria para organizar el procesamiento paralelo. También, este modelo muestra los aspectos asociados a la configuración inicial del contexto de programación basada en "Esqueletos Algoritmicos Reconfigurables", así como la recuperación de los recursos asignados para la terminación adecuada del contexto de computación en la plataforma de computación heterogénea. (Ver Figura 4.6).

```
1 /* Object design model for SkeletonCore Enviorenment */
2 Object Model: skeletoncorePapi {
3     //* Declare data variables
4     int platformID;     // Platform Identification
5     int deviceGroup[3]; // Grouping devices: CPU, GPU and FPGA
6     int cpu_deviceID;   // OpenCL Device Identificador for CPU
7     int gpu_deviceID;   // OpenCL Device Identificador for GPU
8     int fpga_deviceID;  // OpenCL Device Identificador for FPGA
9     //* Declare Operations
10    void init()  { }; // Begin processing enviorenment
11    void terminate() { }; // Finish processing enviorenment
12    void platformDiscovery(int deviceGroup[3]) { };      // Check if OpenCL devices
13    void manageSkeletoncoreError(int msg_err) { }; // Manage errors in Skeletoncore
14    void getOpenCLPlatforms(int &platformID, int &retNumPlatforms) { }; // Get Devices from platform
15 } new mySkelcore enviorement; // for instance
```

Figura 4.6: Segmento de código algorítmico en pseudo C/C++ que muestra el diseño de los aspectos de configuración del ambiente de computación SkeletonCoRe sobre la plataforma heterogénea. Fuente: Elaborado por el autor.

B) Modelo de Código de "Data Set": Este objeto de diseño permite modelar los datos de entrada según su utilidad y awociación a un tipo de problema determinado para su procesamiento. Es posible considerar modelos de datos para contener imágenes, matrices para sistemas de ecuaciones lineales, vectores, etc. (Ver Figura 4.7).

```
/* Object design model for Data container */
Object Model: dataSet {
    //* Declare data variables
    Object Model: matrixObject;
    Object Model: vectorObject;
    Object Model: imageObject {
        //* Declare data variables
        string  *in_imageName[list_size];  // Input images array
        string  *out_imageNamelist_size]; // Output images array
        struct  imageProperties {// Properties of Image
            int imageType;       // Image type of JPG, PNG, BMP, etc (bitmap format)
            int width;           // Columns
            int height;          // Rows
            int quality;         // Image quality for jpg images
            int ColorRGBChannels;// Numbers of RGB Color channels
            int GrayscaleChannel;// Gray channel
            //* Declare Operations
        };
        //* Declare Operations (image Object)
        void readInputImages(string my_imageType, int dimensions,
                string *my_in_imageName[256]) { }; // Read input data
        void writeOutputImages(string *my_out_imageName[]) { }; // Write output data
    } = new my_imageObject; // For instance
     //* Declare Operations (dataSet Object)
    void setupdataObject(model <dataSet:imageObject> my_imageObject) { }; // Configuring the data object
} = new my_dataSet; // For instance
```

Figura 4.7: Segmento de código algorítmico en pseudo C/C++ que muestra el diseño de los objetos de datos del ambiente de computación de SkeletonCoRe.
Fuente: Elaborado por el autor.

C) Modelo de Código de "Device Partition": Aqui se aprovechan varias propiedades del enfoque orientado a objetos como por ejemplo la capacidad de modelar entidades encapsulando su estructura y comportamiento, con sus datos (variables o atributos) y operaciones propias (funciones o métodos), y en consecuencia ocultando los detalles de implementación de éste. Con base en lo anterior se define entonces un objeto "Partition de Computación" asociada a un dispositivo (device partition) en la cual se adopta la noción de partición como un ambiente de computación que contiene y oculta los datos y operaciones (tareas) asociados al contexto de computación del esqueleto. Algunos datos y operaciones de ésta clase (objeto) son: **a) Taskspool:** Conjunto de operaciones o tareas que ejecuta el esqueleto sobre los datos, **b) Channel:** Son los canales que permiten la interacción y comunicación entre tareas en las particiones de los dispositivos, corriendo en el mismo o diferentes contextos de computación, **c) Dataset:** Comprende las estructuras de datos más comunmente usados en aplicaciones de alto rendimiento, como imágenes, matrices, arreglos, etc., y **d) ParType:** Cada partición de Computación está asociada a un dispositivo de procesamiento disponible en la plataforma heterogénea (Ver Figura 4.8).

```
/* Object design model for Device Computing Partition */
Object Model: devicePartition { // Configuration of Partition for CPU, GPU and FPGA
    //* Declare data variables
    int    partitionType;    // Partition type for CPU, GPU and FPGA
    int    deviceID;         // Device Identification for CPU, GPU and FPGA
    int    comm_channel;     // ARRAY of communication channel buffers
    int    host_comm_channel;// Communication  channel to Host
    int    context;          // Execution context for processing
    int    inputMemBuffer;   // Input data objects for processing
    int    outputMemBuffer;  // Output data objects for processing
    string *commandQueue;    // Commands for processing
    string list_inImageNames[list_size]; // List of input images
    string list_outImageNames[list_size];// List of output images
    string tasksPool[3];     // ARRAY of Tasks or Kernels
    FILE   kernelFiles[3];   // ARRAY of Kernel Files
    float  t_ini_device, t_end_device, t_total_device; // Performance: exec time metrics
    //* Declare Operations
    void   setupP(int deviceGroup, model <devicePartition> myPartition, model <dataSet:imageObject> mydataSet,
                string tasksPool) { };                          // Configuring the device Partition
    void   getdataSet(model <dataSet:imageObject> dataObject) { }; // Managing input Data for processing
    void   putdataSet(model <dataSet:imageObject> dataObject) { }; // Managing output Data for processing
    void   getexecTime(float t_init, float t_end, float total_device) { };// Get performance metrics
} = new my_particionCPU; // For instance
```

Figura 4.8: Segmento de código algorítmico en pseudo C/C++ que muestra el diseño de la estructura de una partición de computación de dispositivo. Fuente: Elaborado por el autor.

D) Modelo de Código de "Parallel Skeleton Frame": Esta clase sirve de marco para modelar y diseñar los esqueletos como objetos o funciones de orden superior (high-order functions or high-order objects), donde se aprovecha principalmente la encapsulación, la herencia y el polimorfismo. También, muestra la estructura y comportamiento de un esqueleto para realizar procesamiento paralelo aprovechando los dispositivos de computación disponibles en la Plataforma Heterogénea de Computación Reconfigurable. Esta función se modela y diseña como funciones de orden superior (high-order functions), que toman un objeto "devicePartition" como una partición de Computación que se pasa como argumento, aprovechando para ello las propiedades de encapsulación, la herencia y el polimorfismo del enfoque orientado a objetos (Ver Figura 4.9).

```
/* Object design model for a Computing Parallel Skeleton Frame */
Object Model: skeletonParallel { // Parallel Skeletons for Processing on CPU, GPU and FPGA
    //* Declare data variables
    int   skeletonType; // Skeleton type for Pipe, Master/Slave, SEQ, etc.
    int host_comm_channel;
    //* Declare general operations
    void init_()   { };   // Begin the processing enviorenment
    void finish_() { }; // Terminate the processing enviorenment
    void getdataSet(model <dataSet:imageObject> dataObject) { };  // Manage Input/Output Data for processing
    void putdataSet(model <dataSet:imageObject> dataObject) { };  // Manage Input/Output Data for processing
    void getExecTime(float t_init, float t_end, float t_totalDevice) { };// Get performance metrics
    void createBuffer(int *inputMemBufferCPU, int *inputMemBufferGPU, int *inputMemBufferFPGA) { };
    void setupKernel(string cpu_tasksPool,  FILE *cpu_kernelFiles,
                          string gpu_tasksPool,  FILE *gpu_kernelFiles,
                          string fpga_tasksPool, FILE *fpga_kernelFiles) { };// Manage Input/Output Data for processing
    void connectDevices(int *host_comm_channel, int *cpu_comm_channel, int *gpu_comm_channel, int *fpga_comm_channel) { };
    void executeDevicePartitions(model <devicePartition> CPUpartition, model <devicePartition> GPUpartition,
            model <devicePartition> FPGApartition) { };
    /* Declare the Reconfigurable Parallel Skeletons of SkeletonCoRe:
       Generic Parallel Skeleton: void parallelSkeleton(class devicePartition1, ..., class devicePartitionN);*/
    void generic_Skeleton(model <devicePartition> myCPUpartition, model <devicePartition> myGPUpartition,
            model <devicePartition> myFPGApartition) { };
    void PipeSkeleton(model <devicePartition> CPUpartition, model <devicePartition> GPUpartition,
            model <devicePartition> FPGApartition) { }; // Pipeline Parallel Processing Skeleton
    void TaskSkeleton(model <devicePartition> CPUpartition, model <devicePartition> GPUpartition,
            model <devicePartition> FPGApartition) { }; // Master-Slave Parallel Processing Skeleton
    void SEQSkeleton(model <devicePartition> CPUpartition) { }; // Sequential Processing Skeleton
    void generic_parallelSkeleton(model <devicePartition> myCPUpartition, model <devicePartition>
            myGPUpartition, model <devicePartition> myFPGApartition)
    {
        init_();
        // Creating memory objects (buffers) to hold data
        createBuffer(&myCPUpartition.inputMemBuffer, &myGPUpartition.inputMemBuffer, &myFPGApartition.inputMemBuffer);
        // Get Kernel file
        setupKernel(*myCPUpartition.tasksPool,  myCPUpartition.kernelFiles,
                    *myGPUpartition.tasksPool,  myGPUpartition.kernelFiles,
                    *myFPGApartition.tasksPool, myFPGApartition.kernelFiles);
        // Connecting device channels for Processing
        connectDevices(&host_comm_channel, &myCPUpartition.comm_channel, &myGPUpartition.comm_channel,
                &myFPGApartition.comm_channel);
        // Executing kernel on each device
        executeDevicePartitions(myCPUpartition, myGPUpartition,  myFPGApartition);
        finish_();
    }
} new parallelSkeleton; // for instance
```

Figura 4.9: Segmento de código algorítmico en pseudo C/C++ que muestra la clase con el diseño del marco genérico de un Esqueleto de Computación Reconfigurable. Fuente: Elaborado por el autor.

E) Modelo de Código de "Main Parallel Program": Este código muestra de forma pseudoformal un programa principal que usa el PAPI SkeletonCoRe con los modelos de diseño explicados, mostrando los esqueletos como objetos o funciones de orden superior (high-order functions or high-order objects), donde se el potencial paralelismo de forma implícita. (Ver Figura 4.10).

```
122 /* Example of Main Parallel Program Using the Skeletoncore PAPI */
123 int main(int argc, char *argv[]) {
124     //* Declare data variables
125     int    dimensions = columns * rows;   // width x height, for example
126     string imageType  = "jpg";            // type of jpg, png, etc, for example
127     string *list_inImageNames[list_size]; // Input images queue, for example
128     string *list_outImageNames[list_size];// Output images queue, for example
129     string  mytasksPool[3];
130     //* Declare skeletoncore objects
131     model type: <skeletoncorePapi>    skelcore;        // Create the setup enviorement
132     model type: <dataSet:imageObject> mydataSet;       // Create container for images
133     model type: <devicePartition>     myCPUPartition;  // device Partition for CPU Processing
134     model type: <devicePartition>     myGPUPartition;  // device Partition for GPU Processing
135     model type: <devicePartition>     myFPGAPartition; // device Partition for FPGA Processing
136     model type: <skeletonParallel>    myparallel;
137     //* Init the skeletoncore processing enviorement
138     skelcore.init();
139     //*** Start processing ***
140     //* Getting info of computing devices in platform
141     skelcore.platformDiscovery(skelcore.deviceGroup);
142     //* Getting input data
143     mydataSet.readInputImages(imageType, dimensions, &list_inImageNames[256]);
144     //* Setting Up Partitions for devices
145     myCPUPartition.setupP(skelcore.deviceGroup[0], myCPUPartition,  mydataSet, mytasksPool[0]);
146     myGPUPartition.setupP(skelcore.deviceGroup[1], myGPUPartition,  mydataSet, mytasksPool[1]);
147     myFPGAPartition.setupP(skelcore.deviceGroup[2], myFPGAPartition, mydataSet, mytasksPool[2]);
148     //* Executing parallel skeleton on computing devices
149     myparallel.generic_Skeleton(myCPUPartition, myGPUPartition, myFPGAPartition);
150     //* Getting output data
151     mydataSet.writeOutputImages(*myCPUpartition.list_outImageNames); // Funciona bien
152     //* Finish processing
153     //* Ended the skeletoncore processing enviorement
154     skelcore.terminate();
155     print("---> Main Parallel Program ended susscessfull!\n");
156     return 0;
157 }
```

Figura 4.10: Segmento de código algorítmico en pseudo C/C++ que muestra el diseño de un programa principal usando un esqueleto paralelo genérico de **SkeletonCoRe**. Fuente: Elaborado por el autor.

Como se muestra en las figuras anteriores, el enfoque basado en objetos ha permitido modelar y diseñar los esqueletos como clases y funciones de orden superior. En la presente investigación, ha sido posible el diseño e implementación de "Esqueletos Algorítmicos Reconfigurables de Orden Superior" mostrando una super-estructura que se has denominado "Particiones de Computación" por tipo de dispositivo de cómputo, las cuales son pasadas como parámetro a los esqueletos. Esta estructura modela objetos contenedores de datos, de configuración y procesamiento (tareas) en términos del comportamiento de un contexto de computacion asociado a un dispositivo OpenCL de procesamiento particular.

Es así, como se pudo modelar de forma genérica los objetos generales de SkeletonCoRe y aplicar el concepto de funciones de orden superior **(propiedad 3: función que toma como argumento a otra función, subsección 3.4.5.2)** han permitido diseñar e implementar los esqueletos como "objetos de orden superior".

Como un apoyo adicional, en la siguiente subsección se describen de forma algoritmica el comportamiento de los esqueletos paralelos reconfigurables que son implementados y probados en la presente tesis doctoral.

4.4.2 *DISEÑO DE LOS ALGORÍTMICO DE LOS ESQUELETOS RECONFIGURABLES*

En la subsección anterior se han definido los modelos de clases y objetos de los elementos que conforman el **PAPI SkeletonCore**. Ahora, en la presente subección se definen los algoritmos, en código pseudoformal, que muestran la estructura y comportamiento de los esqueletos paralelos reconfigurables y un programa que muestra el uso de "SkeletonCoRe (Núcleo de Esqueletos Algorítmicos Reconfigurables)", es decir, las instrucciones y esqueletos paralelos asociados a la Interfaz de Programación Implícita de Aplicaciones Paralelas (PAPI) **SkeletonCoRe** (Ver Algorítmos 4.1, 4.2, 4.3 y 4.4).

A contiuación se describe mediante el algoritmo 4.1 los datos, operaciones auxiliares y los esqueletos paralelos reconfigurables que se pueden usarse en un contexto general de la PAPI SkeletonCoRe:

Algoritmo 4.1: Algorítmo genérico pseudoformal que muestra la Estructura de un Programa Paralelo Usando la sintaxis y semántica de las instrucciones de la PAPI *SkeletonCoRe*. Fuente: Elaborado por el Autor.

```
Library    : Open <skeletoncore.lib>
InputData  : ImageFile, Vector, Matrix; such that a bitmap ImageFile of size NxN
OutputData: ImageFile, Vector, Matrix; such that a bitmap ImageFile of size NxN
Begin_Main
    /* Create a Skeletoncore Environment */
    Skeletoncore.create(skelcore)
    /* Get the environment variables for the SkeletonCoRe PAPI */
    skelcore.init()
        /* Discovery of Platform OpenCL Devices */
        skelcore.getOpenCLDevices(deviceID[0...2])
        if deviceID[] ≠ NULL, with deviceID[] ∈ {CPU, GPU, FPGA, NULL} then
            /* Get the input Data and Dimension for Processing */
            skelcore.readData(input imageFile, Vector, Matrix, sizeData)
            /* Get the Computing Kernels */
            skelcore.getKernels(kernelTask[0...2])
            for device = 0, where deviceID[device] ∈ {CPU, GPU, FPGA} do
                /* Create the Device Partition and get the Parameters */
                skelcore.Partition[device](
                                deviceID[device], /* Device type assigned to Partition */
                                kernelTask[device], /* Computing Kernel assigned to Device */
                                inputData, /* Input Data for Processing */
                                ouputData, /* Ouput Data Processed */
                                sizeData, /* Dimension/Size of the Data */
                                perfMetrics) /*Processing Metrics of Data Processed */
            end
            /* User's input of devices Group for design space exploration */
            skelcore.getExploringOption(devicePool)
            /* Parallel Template for Processing Using the Pipeline Skeleton */
            while devicesGroup ≠ NULL, where devicesGroup ∈ {{CPU}, {GPU}, {FPGA},
              {CPU, GPU}, {CPU, FPGA}, {GPU, FPGA}, {CPU, GPU, FPGA}} do
                /* Exploring Design Spaces Using the Pipeline Skeleton */
                skelcore.PipeSkeleton(skelcore.Partition[], devicesGroup, execMetrics)
                /* Write result and Performance Metrics in output file */
                skelcore.writeResult(output{imageFile, Vector, Matrix}, execMetrics)
                /* Exploring Design Spaces Using the Master/Slave Skeleton */
                skelcore.TaskSkeleton(skelcore.Partition[], devicesGroup, execMetrics)
                /* Write result and Performance Metrics in output file */
                skelcore.writeResult(output{imageFile, Vector, Matrix}, execMetrics)
                /* A New User's input of devices Pool for design space exploration */
                skelcore.getExploringOption(devicePool)
            end
        else if deviceID[] = NULL then
            /* No Computing Devices Detected on this Platform */
        end
    end
    /* Exploring Processing Using the SEQuential Skeleton with no OpenCL */
    skelcore.SEQSkeleton(skelcore.Partition[CPU], execMetrics)
    /* Write result and Performance Metrics in output file */
    skelcore.writeResult(output{imageFile, Vector, Matrix}, execMetrics)
    /* SkeletonCoRe Ending: Kill Processes and Free Resources */
    skelcore.terminate()
End_Main
```

Así, como en el algoritmo anterior, aquí se puede observar el siguiente algoritmo 4.2 que describe de foma genérica los datos, parámetros, estructura y comportamiento interno (encapsulado) del esqueleto paralelo **PipeSkeleton**:

Algoritmo 4.2: Algorítmo genérico pseudoformal del Esqueleto paralelo PipeSkeleton. Fuente: Elaborado por el Autor.

```
Input  : array struct Partition[devicesGroup], where devicesGroup ∈ {{CPU}, {GPU}, {FPGA}, {CPU, GPU},
         {CPU, FPGA}, {GPU, FPGA}, {CPU, GPU, FPGA}; InputFile ∈ {imageFile, Vector, Matrix}; such that a
         bitmap ImageFile of size NxN.
Output: Output ∈ {imageFile, Vector, Matrix}; such that a bitmap ImageFile of size NxN, and Metrics for
         performance metrics.
1  Function PipeSkeleton(array struct Partition[], set devicesGroup, file Output, real Metrics):
2      /* Configuring the Pipeline Parallelism */
3      foreach devicesGroup ∈ {{CPU}, {GPU}, {FPGA}, {CPU, GPU}, {CPU, FPGA},
4                                                {GPU, FPGA}, {CPU, GPU, FPGA}} do
5          if devicesGroup = {CPU} then
6              /* Only CPU Activated for Computing */
7              startExec(Partition[CPU].kernelTask, Partition[CPU].inputData,
8                        Partition[CPU].outputData, Partition[CPU].perMetrics)
9              Result ⟵ Partition[CPU].outputData
10         else if devicesGroup = {GPU} then
11             /* Only GPU Activated for Computing */
12             startExec(Partition[GPU].kernelTask, Partition[GPU].inputData,
13                       Partition[GPU].outputData, Partition[GPU].perMetrics)
               Result ⟵ Partition[GPU].outputData
14         else if devicesGroup = {FPGA} then
15             /* Only FPGA Activated for Computing */
16             startExec(Partition[FPGA].kernelTask, Partition[FPGA].inputData,
17                       Partition[FPGA].outputData, Partition[FPGA].perMetrics)
               Result ⟵ Partition[FPGA].outputData
18         else if devicesGroup = {CPU, GPU} then
19             /* CPU and GPU Activated for Computing */
20             processCPU ⟵ {Partition[CPU].kernelTask, Partition[CPU].inputData,
21                       Partition[CPU].outputData, Partition[CPU].perMetrics}
22             processGPU ⟵ {Partition[GPU].kernelTask, Partition[GPU].inputData,
23                       Partition[GPU].outputData, Partition[GPU].perMetrics}
24             /* Create connection channel for CPU and GPU */
25             pipeChannelCPU-GPU ⟵ connectDevices(ProcessCPU, ProcessGPU)
26             /* Start the Parallel Execution CPU and GPU */
27             startParallExec(ProcessCPU, pipeChannelCPU − GPU, ProcessGPU)
               Result ⟵ Partition[GPU].outputData
28         else if devicesGroup = {CPU, FPGA} then
29             /* CPU and FPGA Activated for Computing */
               .
               .
               .
30         else if devicesGroup = {GPU, FPGA} then
31             /* GPU and FPGA Activated for Computing */
               .
               .
               .
32         else if devicesGroup = {CPU, GPU, FPGA} then
33             /* CPU, GPU and FPGA Activated for Computing */
34             processCPU ⟵ {Partition[CPU].kernelTask, Partition[CPU].inputData,
35                       Partition[CPU].outputData, Partition[CPU].perMetrics}
36             processGPU ⟵ {Partition[GPU].kernelTask, Partition[GPU].inputData,
37                       Partition[GPU].outputData, Partition[GPU].perMetrics}
38             processFPGA ⟵ {Partition[FPGA].kernelTask, Partition[FPGA].inputData,
39                       Partition[FPGA].outputData, Partition[FPGA].perMetrics}
40             /* Create pipe connection channels for CPU, GPU and FPGA */
41             pipeChannelCPU-GPU-FPGA ⟵ connectDevices(ProcessCPU, ProcessGPU)
42             pipeChannelGPU-FPGA ⟵ connectDevices(ProcessGPU, ProcessFPGA)
43             /* Start the Parallel Execution CPU, GPU and FPGA */
44             startParalelExec(ProcessCPU, pipeChannelCPU-GPU, ProcessGPU, pipeChannelGPU-FPGA,
45                              ProcessFPGA)
46             /* Get result from the last stage of Pipeline */
47             Result = Partition[FPGA].outputData
48         end
49     end
50     return Result
51 End Function
```

Por último, el algoritmo 4.3 muestra los datos, operaciones, estructura y comportamiento general del esqueleto paralelo **TaskSkeleton**:

Algoritmo 4.3: Algorítmo genérico pseudoformal del Esqueleto TaskSkeleton. Fuente: Elaborado por el Autor.

```
Input  : array struct Partition[devicesGroup], where devicesGroup ∈ {{CPU}, {GPU}, {FPGA}, {CPU, GPU},
         {CPU, FPGA}, {GPU, FPGA}, {CPU, GPU, FPGA}; InputFile ∈ {imageFile, Vector, Matrix}; such that a
         bitmap ImageFile of size NxN.
Output: Output ∈ {imageFile, Vector, Matrix}; such that a bitmap ImageFile of size NxN, and Metrics for
         performance metrics.
1  Function TaskSkeleton(array struct Partition[], set devicesGroup, file Output, real Metrics):
2      /* Configuring the Master/Slave Parallelism */
3      foreach devicesGroup ∈ {{CPU}, {GPU}, {FPGA}, {CPU, GPU}, {CPU, FPGA},
4                              {GPU, FPGA}, {CPU, GPU, FPGA}} do
5          if devicesGroup = {CPU} then
6              /* Only CPU Activated for Computing */
7              startExec(Partition[CPU].kernelTask, Partition[CPU].inputData,
8                        Partition[CPU].outputData, Partition[CPU].perMetrics)
9              Result ⟵ Partition[CPU].outputData
10         else if devicesGroup = {GPU} then
11             /* Only GPU Activated for Computing */
12             startExec(Partition[GPU].kernelTask, Partition[GPU].inputData,
13                       Partition[GPU].outputData, Partition[GPU].perMetrics)
               Result ⟵ Partition[GPU].outputData
14         else if devicesGroup = {FPGA} then
15             /* Only FPGA Activated for Computing */
16             startExec(Partition[FPGA].kernelTask, Partition[FPGA].inputData,
17                       Partition[FPGA].outputData, Partition[FPGA].perMetrics)
               Result ⟵ Partition[FPGA].outputData
18         else if devicesGroup = {CPU, GPU} then
19             /* CPU and GPU Activated for Computing */
20             processCPU ⟵ {Partition[CPU].kernelTask, Partition[CPU].inputData,
21                       Partition[CPU].outputData, Partition[CPU].perMetrics}
22             processGPU ⟵ {Partition[GPU].kernelTask, Partition[GPU].inputData,
23                       Partition[GPU].outputData, Partition[GPU].perMetrics}
24             /* Create connection channels for CPU and GPU slaves to CPU Master */
25             pipeChannelCPU-CPU ⟵ connectDevices(ProcessCPU, ProcessCPU)
26             pipeChannelCPU-GPU ⟵ connectDevices(ProcessCPU, ProcessGPU)
27             /* Start the Parallel Execution CPU and GPU */
28             startParallExec(pipeChannelCPU-CPU, ProcessCPU, pipeChannelCPU-GPU, ProcessGPU)
29             /* CPU Master integrates the sub results from CPU and GPU */
30             Result ⟵ integrateResult(Partition[CPU].outputData, Partition[GPU].outputData)
31         else if devicesGroup = {CPU, FPGA} then
32             /* CPU and FPGA Activated for Computing */
               .
               .
33         else if devicesGroup = {GPU, FPGA} then
34             /* GPU and FPGA Activated for Computing */
               .
               .
35         else if devicesGroup = {CPU, GPU, FPGA} then
36             /* CPU, GPU and FPGA Activated for Computing */
37             processCPU ⟵ {Partition[CPU].kernelTask, Partition[CPU].inputData,
38                       Partition[CPU].outputData, Partition[CPU].perMetrics}
39             processGPU ⟵ {Partition[GPU].kernelTask, Partition[GPU].inputData,
40                       Partition[GPU].outputData, Partition[GPU].perMetrics}
41             processFPGA ⟵ {Partition[FPGA].kernelTask, Partition[FPGA].inputData,
42                       Partition[FPGA].outputData, Partition[FPGA].perMetrics}
43             /* Create connection channels for CPU-CPU, CPU-GPU and CPU-FPGA */
44             pipeChannelCPU-CPU ⟵ connectDevices(ProcessCPU, ProcessCPU)
45             pipeChannelCPU-GPU ⟵ connectDevices(ProcessCPU, ProcessGPU)
46             pipeChannelCPU-FPGA ⟵ connectDevices(ProcessCPU, ProcessFPGA)
47             /* Start the Parallel Execution CPU, GPU and FPGA */
48             startParalelExec(pipeChannelCPU-CPU, ProcessCPU, pipeChannelCPU-GPU, ProcessGPU,
49                              pipeChannelCPU-FPGA, ProcessFPGA)
50             /* CPU Master integrates the sub results from CPU, GPU and FPGA */
51             Result ⟵ integrateResult(Partition[CPU].outputData, Partition[GPU].outputData, Partition[FPGA].outputData)
52         end
53     end
54     return Result
55 End Function
```

Como un caso especial, el algoritmo 4.4 muestra la estructura y comportamiento del esqueleto secuencial **SEQSkeleton** usado para encapsular tareas secuenciales:

Algoritmo 4.4: Algorítmo genérico pseudoformal del Esqueleto Secuencial SEQSkeleton. Fuente: Elaborado por el Autor.

```
Input  : struct PartitionCPU, InputFile ∈ {imageFile, Vector, Matrix}; such that a bitmap ImageFile of size NxN.
Output: Output ∈ {imageFile, Vector, Matrix}; such that a bitmap ImageFile of size NxN, and Metrics for
        performance metrics.
1 Function SEQSkeleton(struct PartitionCPU, file Output, real Metrics):
2     /* Configuring the Sequential Processing with no OpenCL */
3     /* Only CPU Activated for Computing Using Just one Thread or Process */
4     startExec(PartitionCPU.kernelTask, PartitionCPU.inputData, PartitionCPU.outputData,
5               PartitionCPU.perMetrics)
6     Result = PartitionCPU.outputData
7     return Result
8 End Function
```

En ésta sección se han descrito las características de los elementos involucrados en los esqueletos reconfigurables de **SkeletonCoRe** que serán implementados en OpenCL C/C++. Estos esqueletos, **PipeSkeleton** y **TaskSkeleton**, se describen con detalle en los capítulos 5 y 6, respectivamente, del presente documento, asímismo los códigos fuente en C/C++ y en OpenCL C/C++ estan disponible en la sección del Anexo C, al final del documento.

Un caso especial que se debe mencionar es el Esqueleto **SEQSkeleton** el cual, aunque no es un esqueleto paralelo, sirve de esqueleto trivial para encapsular código secuencial en C/C++ y usarlo en cualquier aplicación que use el PAPI **SkeletonCoRe**.

4.5 DESARROLLO DE LOS ESQUELETOS RECONFIGURABLES DE SKELETONCORE

4.5.1 *DESCRIPCIÓN DE LA PLATAFORMA DE PROGRAMACIÓN DE* SKELETONCORE

Para la evaluación de nuestra herramienta hemos empleado una plataforma de computación heterogénea reconfigurable conformada por una placa base AsRock que soporta un procesador de décima generación Intel® Core™ 10ma Gen i5-10400F de 6 núcleos (12 hilos) a 2.9 Ghz y dos bancos de memoria DDR4 de 8 GB cada uno a 2933 Mhz. Dicha tarjeta base está equipada con una tarjeta FPGA Intel® Stratix® 10 GX con 10.2 millones de elementos lógicos (LEs, Logic Elements) y una tarjeta gráfica NVIDIA GeForce GTX 1060 con 1.280 núcleos y 6 GB de memoria. Todos estos dipositivos están conectados a través de un bus de sistema de 64 bits con un ancho de banda de 2.9 Gbits/segundo.

Además, el sistema de computación heterogéneo utiliza como sistema operativo la distribución Linux Ubuntu versión 20.04.4 LTS, y el paquete de compiladres de GNU GCC versión 9.4.0 para compilar los códigos del Host (CPU-GPU) y del FPGA en lenguaje openCL versión 2.1.

Como soporte a todo lo anterior, se usa el ambiente de desarrollo de aplicaciones para sistemas heterogéneos denominado Intel Quartus Prime Design Software versión 22.1.0 con Intel FPGA SDK para OpenCL versión 2.1 para Linux. Con base en la arquitectura de la PAPI SkeletonCoRe hemos elegido este ambiente de desarrollo (SDK, Software Development Kit) por cuanto esta herramienta integra herramientas que cubren las cuatro capas inferiores de la arquitectura de SkeletonCoRe (las capas superiores están cubiertas por el API OpenCL C/C++), y por tanto facilitan la construcción de las funcionalidades asociadas al proceso de compilacion de los componentes para el CPU y GPU (scheduling) y la síntesis (partition, placement y routing) de los componentes o tareas para el FPGA.

Para una descripción más detallada de la plataforma de hardware (CPU, GPU y FPGA) y del ambiente de diseño y programación Intel FPGA SDK para OpenCL se aconseja referirse a la sección respectiva en el anexo B. También, como complemento a la arquitectura antes descrita, en la anterior Fig. 4.4b) se puede observar un diagrama que refleja las actividades asociadas en el proceso de Codiseño y Programación usuando SkeletonCoRe.

4.5.2 *INSTALACIÓN, CONFIGURACIÓN Y PRUEBA DE LA PLATAFORMA DE DESARROLLO*

En esta subsección se muestra la instalación y configuración del API OpenCL. Para ello, se tiene una plataforma heterogénea de computación ya descrita y un sistema operativo basado en la distribución Linux Ubuntu LTS 20.04, así como un editor de programación como Visual Studio Code o un ambiente de desarrollo con varias heramientas integradas como QUARTUS Prime version 21.1.0.

El proceso de preparación del ambiente para codificar, compilar y correr programas implementados en OpenCL C/C++ es el siguiente:

Primer paso: Instalación del compilador Gnu C/C++, el depurador Git, el script CMake, la librería y drivers del CPU, GPU y del FPGA con soporte **OpenCL**.

```
> sudo apt update; apt -y upgrade
> sudo apt install build-essential -y
> sudo apt install git -y
> sudo apt install cmake -y
> sudo apt install clinfo opencl-headers ocl-icd-opencl-dev -y
> clinfo -l
> sudo reboot
```

Otra alternativa para la preparación del entorno de programación en OpenCL/C++ en Linux Ubuntu LTS 20.04 es el siguiente:

```
> mkdir neo
> cd neo
> wget https://github.com/intel/intel-graphics-compiler/releases/download/igc-1.0.12149.1 \
intel-igc-core_1.0.12149.1_amd64.deb
> wget https://github.com/intel/intel-graphics-compiler/releases/download/igc-1.0.12149.1 \
intel-igc-opencl_1.0.12149.1_amd64.deb
> wget https://github.com/intel/compute-runtime/releases/download/22.39.24347 \
intel-level-zero-gpu-dbgsym_1.3.24347_amd64.ddeb
> wget https://github.com/intel/compute-runtime/releases/download/22.39.24347 \
intel-level-zero-gpu_1.3.24347_amd64.deb
> wget https://github.com/intel/compute-runtime/releases/download/22.39.24347 \
intel-opencl-icd-dbgsym_22.39.24347_amd64.ddeb
> wget https://github.com/intel/compute-runtime/releases/download/22.39.24347 \
intel-opencl-icd_22.39.24347_amd64.deb
> wget https://github.com/intel/compute-runtime/releases/download/22.39.24347/libigdgmm12_22.2.0_amd64.deb
> sudo apt install ./*.deb
> /usr/bin/clinfo -l
> sudo reboot
```

En referencia a la instalación del entorno de programación OpenCL no hay que olvidar que es necesario instalar los drivers OpenCL del CPU, de la tarjeta GPU y de la tarjeta FPGA según la marca (plataforma) y modelo de cada una.

Segundo paso: Instalación del editor de programación Visual Studio Code y el ambiente de desarrollo Intel Quartus Prime Design Software versión 22.1.0 con el Kit de Intel FPGA SDK para OpenCL

versión 2.1 para Linux. En este caso también se puede instalar las librerías de Intel oneAPI. Para obtener el software e instrucciones de instalación se debe visitar el enlace al portal de Intel Co. https://www.intel.la/content/www/xl/es/support/programmable/support-resources/design-examples/horizontal/opencl.html.

Para probar la instalación del entorno de programación OpenCL compile y corra el programa ejemplo mostrado en el código fuente 3.8 del Capítulo 3, el cual suma dos (2) vectores de números en punto flotante, usando CPU y GPU.

4.5.3 *ELEMENTOS DEL API OPENCL/C++ USADOS PARA LA EXPRESIÓN DE PARALELISMO*

Como continuación a la explicación del API OpenCL, en esta subsección se describen los elementos principales del API que permiten expresar y explotar paralelismo en un programa.

Como se ha mencionado, en el modelo OpenCL una plataforma heterogénea se compone de una unidad central de procesamiento (denominada Host) y varios dispositivos de procesamiento especializados. Según el modelo de plataforma de OpenCL, el host son los procesadores de propósito general con arquitectura x86, AMD o similares. Mientras que los dispositivos de cómputo GPU, DSP y FPGA son los procesadores especializados. En este sentido dependiendo de las implementaciones, los núcleos (cores) pueden ser considerados de diversas granularidades.

Los elementos esenciales y principales que componen la estructura básica para la construcción de un programa OpenCL son:

1. Discovery Platform: Para descubrir la plataforma heterogénea con OpenCL se debe usar la siguiente función que permite determinar la cantidad de plataformas:

```
cl_int  clGetPlatformIDs (cl_uint num_entries,
                          cl_platform_id *platforms,
                          cl_uint *num_platforms)
```

2. Discovery Devices: Una vez que determinamos y seleccionamos la plataforma, consultamos por los dispositivos disponibles en la plataforma. Se puede especificar el tipo de dispositivo que estamos buscando: todos, solo GPUs, solo CPUs, solo FPGAs. La expresión en OpenCL es:

```
clGetDeviceIDs[4] (cl_platform_id platform,
                   cl_device_type device_type,
                   cl_uint num_entries,
                   cl_device_id *devices,
                   cl_uint *num_devices)
```

3. Create Context: Para programar una aplicación en la plataforma es necesario crear un contexto, el cual es un espacio para manejar los objetos y recursos de OpenCL. En un programa OpenCL los siguientes elementos están asociados a un contexto: a) Dispositivos, b) Objetos de programa: implementación de los objetos de cómputo, c) Kernels: Funciones que corren en dispositivos OpenCL (el código de los threads), d) Objetos de memoria: los datos operados en el dispositivo (los datos de los threads), y e) Colas de comandos: mecanismos de interacción de dispositivos (transferencia de datos, ejecución de kernels y sincronización). Con esta función creamos un contexto, se debe pasar la lista de dispositivos çl_context_properties.[es]pecifica que plataforma a usar (NULL indica que se usa el provisto por el vendedor por defecto). Se provee un mecanismo de callback para reportar errores al usuario. La instrucción asociada es:

```
cl_context      clCreateContext (const cl_context_properties *properties,
                                 cl_uint num_devices,
                                 const cl_device_id *devices,
                                 void (CL_CALLBACK *pfn_notify)(const char *errinfo,
                                                const void *private_info, size_t cb,
                                                void *user_data),
                                 void *user_data,
                                 cl_int *errcode_ret)
```

4. Command Queue: La cola de comandos es el mecanismo por el cual el procesador central (host) le pide una acción a un dispositivo (device). Hay transferencia de datos a la memoria y ejecución de tarea. Cada dispositivo tiene su cola de comandos, los cuales pueden ser sincrónicos o asincrónicos. Los comandos se pueden ejecutar en orden o no (out-of-order). Las colas de comandos asocian a los dispositivos con el contexto. En las propiedades se especifica la ejecución fuera de orden y el sensado de desempeño (profilling):

```
cl_command_queue   clCreateCommandQueue (cl_context context,
                                         cl_device_id device,
                                         cl_command_queue_properties properties,
                                         cl_int *errcode_ret)
```

5. Memory Buffer: Los objetos de memoria es la forma de manejar los datos. Se clasifican en buffers o imágenes: a) Buffers: Trozos de memoria contíguos (arreglos, punteros, estructuras), b) Imágenes: Objetos 2D o 3D. Solo se acceden como read_image() y write_image(). Por cada kernel solo se puede leer o escribir. Con esta función creamos un buffer para un contexto dado. Con los flags especificamos: a) La combinación de lectura/escritura permitida en los datos, b) El uso de host pinter para guardar los datos, y c) La copia de los datos desde el host pointer.

```
cl_mem   clCreateBuffer (cl_context context,
                         cl_mem_flags flags,
                         size_t size,
                         void *host_ptr,
                         cl_int *errcode_ret)
```

6. Memory Object: Los objetos de memoria están asociados con un contexto. Deben ser explícitamente transferidos a los dispositivos antes de realizar la ejecución. Los comandos para transferir hacia y desde los dispositivos son: a) clEnqueueRead/WriteBuffer/Image, b) Copiando datos desde el anfitrión (host) al dispositivo (device) es una escritura, c) Copiar desde el dispositivo al host es una lectura. Los objetos de memoria se transfieren a los dispositivos especificando una acción (lectura/escritura) y una cola de comandos:

```
cl_int  clEnqueueWriteBuffer (cl_command_queue command_queue,
                              cl_mem buffer,
                              cl_bool blocking_write,
                              size_t offset,
                              size_t cb,
                              const void *ptr,
                              cl_uint num_events_in_wait_list,
                              const cl_event *event_wait_list,
                              cl_event *event)
```

7. Program Object: En terminos generales, un objeto de programa es una colección de Kernels OpenCL. Puede ser código fuente (texto) o un binario compilado previamente. Puede contener datos constantes o funciones auxiliares. Para crear un objeto programa se requiere leer un archivo de texto (código fuente) o un

binario compilado. Para compilar son necesarios los siguientes requerimientos: a) Especificar el dispositivo destino (hay compilación para cada dispositivo), b) Pasar parámetros al compilador (opcional), y c) Manejar errores de compilación (opcional). Un objeto de programa es creado y compilado cuando se provee los fuentes o un binario. La siguiente función crea un objeto programa usando un archivo de texto conteniendo fuentes:

```
cl_program  clCreateProgramWithSource (cl_context context,
                                       cl_uint count,
                                       const char **strings,
                                       const size_t *lengths,
                                       cl_int *errcode_ret)
```

8. Build Program: Esta función compila y linkea un ejecutable desde el objeto programa a cada dispositivo en el ambiente. En caso de proveer una lista de dispositivos, solo se envía el ejecutable a los dispositivos en la lista.

```
cl_int  clBuildProgram (cl_program program,
                        cl_uint num_devices,
                        const cl_device_id *device_list,
                        const char *options,
                        void (CL_CALLBACK *pfn_notify)(cl_program program,
                                                       void *user_data),
                        void *user_data)
```

9. Create Kernel: Un kernel es una función declarada en un programa que es ejecutada en un dispositivo OpenCL. Un objeto kernel esta compuesto por la función y los argumentos asociados. Un objeto kernel es creado desde un programa compilado. El programa debe asociar explícitamente argumentos (objetos de memoria y primitivas entre otras) con el objeto kernel. Esta función crea un kernel a partir de un objeto programa dado. El objeto kernel creado es especificado por una cadena de caracteres que coincide con el nombre de la función dentro del programa:

```
cl_kernel  clCreateKernel (cl_program program,
                           const char *kernel_name,
                           cl_int *errcode_ret)
```

10. Execution Kernel: La función le dice al dispositivo asociado con una cola de comandos que comience a ejecutar el kernel. El espacio global debe ser especificado y el tamaño de los grupos de trabajo local es opcional. Se puede proveer una lista de eventos que deben cumplirse antes que la operación se ejecute:

```
cl_int  clEnqueueNDRangeKernel (cl_command_queue command_queue,
                                cl_kernel kernel,
                                cl_uint work_dim,
                                const size_t *global_work_offset,
                                const size_t *global_work_size,
                                const size_t *local_work_size,
                                cl_uint num_events_in_wait_list,
                                const cl_event *event_wait_list,
                                cl_event *event)
```

11. Read Buffer: El último paso es copiar los datos desde el dispositivo de cómputo al CPU (Host). Similar a la escritura de datos, pero en este caso hay transferencia del dispositivo al CPU:

```
cl_int clEnqueueReadBuffer (cl_command_queue command_queue,
                            cl_mem buffer,
                            cl_bool blocking_read,
                            size_t offset,
                            size_t cb,
                            void *ptr,
                            cl_uint num_events_in_wait_list,
                            const cl_event *event_wait_list,
                            cl_event *event)
```

Para culminar, al final de un programa OpenCl, la mayoría de los objetos OpenCL deben ser liberados luego de ser usados. Hay una función clReleaseRecurso para la mayoría de los tipos OpenCL. Algunos ejemplos son clReleaseProgramm() o clReleaseMemObject().

4.6 DISEÑO DE PRUEBAS Y EVALUACIÓN DE LOS ESQUELETOS DE SKELETONCORE

4.6.1 *DESCRIPCIÓN DEL EXPERIMENTO DE EVALUACIÓN DE* SKELETONCORE

Las pruebas de los esqueletos van dirigidas a demostrar la abtsracción que se obtiene, la funcionalidad que se mantiene y el rendimiento que se mejora con el uso de los Esqueletos Reconfigurables. Para ello, se usan los siguientes criterios generales:

A) Prueba de Abstracción, dirigida a demostrar que las técnicas utilizadas de programación orientada a objetos y funciones de orden superior efectivamente encapsula y oculta los detalles de bajo nivel explícitos del paralelismo y diseño del FPGA, dejando en evidencia la diferencia de complejidad entre los códigos con y sin esqueletos reconfigurables implementados.

B) Prueba de Funcionalidad, orientada a probar que los esqueletos algorítmicos mantienen el mismo comportamiento de proceso de datos que el código que explota el paralelismo de forma expresa, además generando los mismos resultados que el programa secuencial de prueba sin usar esqueletos ni OpenCL, aunque con tiempos de ejecución diferentes.

C) Prueba de Rendimiento, que busca demostrar que la programación paralela implícita provista por los esqueletos reconfigurables sigue siendo una herramienta que mejora significativamente los tiempos de ejecución con respesto a la solución secuencial.

D) Aplicación que explota paralelismo de datos y de control, utilizando varias tareas de procesamiento intensivo de datos, pero con tareas de diferentes complejidades algoritmicas para obtener distintos tiempos de procesamiento con cada dispositivo de cómputo (CPU, GPU o FPGA). Se implementa una aplicación para procesamiento de imágenes con tres operadores comunes de imagen, como los siguientes: “Color to Grayscale Conversion”, “Sobel’s Edge Detection” e “Image Rotation”. Entonces, se usa como prueba de concepto una aplicación con estos algorítmos de procesamiento de imágenes para evaluar la funcionalidad, abstracción y rendicmiento de todos los Esqueletos Algorítmicos Reconfigurables de la PAPI **SkeletonCoRe**.

F) Evaluación homogénea de todos los esqueletos, *PipeSkeleton y TaskSkeleton* de la PAPI **SkeletonCoRe**, incluyendo el esqueleto secuencial *SEQSkeleton* donde se aplican los operadores o kernels a una cadena o lista de imágenes con diferentes resoluciones o tamaños. Con el fin de reducir el número de configuraciones CPU-GPU-FPGA para exploración de espacios de diseño se hacen mediciones separadas de los tiempos de consumo por procesamiento con CPU, GPU y FPGA; de comunicación entre tareas y de entrada/salida para cada tarea. De esta forma se obtienen las métricas de CPU, GPU y FPGA de cada una de las tareas reduciendo las pruebas a sólo cuatro(4) configuraciones.

F) Se fija un orden de precedencia de estos operadores de imagen (como un pipeline lineal en el caso del esqueleto PipeSkeleton o un árbol master/slave de un nivel en el caso del esqueleto TaskSkeleton) en la forma siguiente: → Color to Grayscale Conversion → Sobel's Edge Detection → Image Rotation.

G) Carga de trabajo de prueba, compuesta de cuatro listas de 256 imagenes, c/u con diferentes dimensiones y tamaños: 256x256, 512x512, 1024x1024 y 2048x2048 pixeles, respectivamente, para explotar paralelismo de datos y de tareas, con procesamiento intensivo en el caso de los esqueletos paralelos, **siendo cada imagen una unidad individual de procesamiento, sin segmentar**.

H) Configuraciones de prueba, se han planificado particiones con asignación de las tareas según el modelo de computo paralelo del Esqueleto Reconfigurable en la forma CPU-GPU-FPGA, siendo tres (3) las tareas, cada una aplicando un operador de imagen diferente, "Color to Grayscale Conversion", "Sobel's Edge Detection" e "Image Rotation". Esto permite obtener las métricas de las tareas en cada configuración, es decir, las tres (3) tareas corriendo sólo en el dispositivo CPU, luego sólo en el dispositivo GPU, después en el dispositivo FPGA, y finalmente una sola tarea en cada dispositivo. Así, se obtienen los tiempos de procesamiento y de comunicación de cada tarea de forma individual en cada dispostivo de cómputo. De esta manera no es necesario probar todas las combianciones de asignaciones posibles de las tareas en CPU-GPU-FPGA.

Por ejemplo, la configuración A con CPU-GPU-FPGA definida como 3-0-0 significa que las tres tareas de procesamiento de imagen estan mapeadas o asignadas sólo a la partición del CPU, dejando libres con 0 tareas las particiones del GPU y el FPGA. Estas configuraciones de cómputo heterogéneo están orientadas a representar diferentes espacios de diseño y rendimiento usando los Esqueletos Algoritmicos Reconfigurables, y se han dispuesto así:

- **Configuración A (CPU-0-0):** Se han mapeado o proyectado todas las tareas de cómputo sólo de software en CPU (3-0-0, cpu-0-0), ver las Figuras 5.2a y 6.2a,
- **Configuración B (0-GPU-0):** Luego, se han mapeado sólo en GPU (0-3-0, 0-gpu-0), ver las Figuras 5.2b y 6.2b,
- **Configuración C (0-0-FPGA):** Después, se ha mapeado todo el pipeline como tareas sólo de hardware en el dispositivo FPGA (0-0-3, 0-0-fpga), ver las Figuras 5.2c y 6.2c.
- **Configuración D (CPU-GPU-FPGA):** Por último, se ha mapeado el pipeline en CPU-GPU-FPGA (1-1-1, cpu-gpu-fpga) dividiéndolo en una etapa o kernel por cada dispositivo de cómputo o etapa, como se puede ver las Figuras 5.2d y 6.2d,

I) Tareas de lectura y escritura de datos, mapeadas de manera fija sólo en el Host (CPU), en todas las configuraciones de prueba. Ésta aplicación de procesamiento de imágenes se usa para la implementación, medición y comparación de las métricas de rendimiento entre los esqueletos paralelos y el secuencial.

En la Figura 4.11 se puede ver el diagrama lógico de la aplicación de procesamiento de imágenes para la prueba de concepto de todos los Esqueletos Algorítmicos Reconfigurables configurada por encauzamiento en línea (pipeline lineal, Figura 4.11a) y maestro-esclavo (master/slave de un nivel, Figura 4.11b).

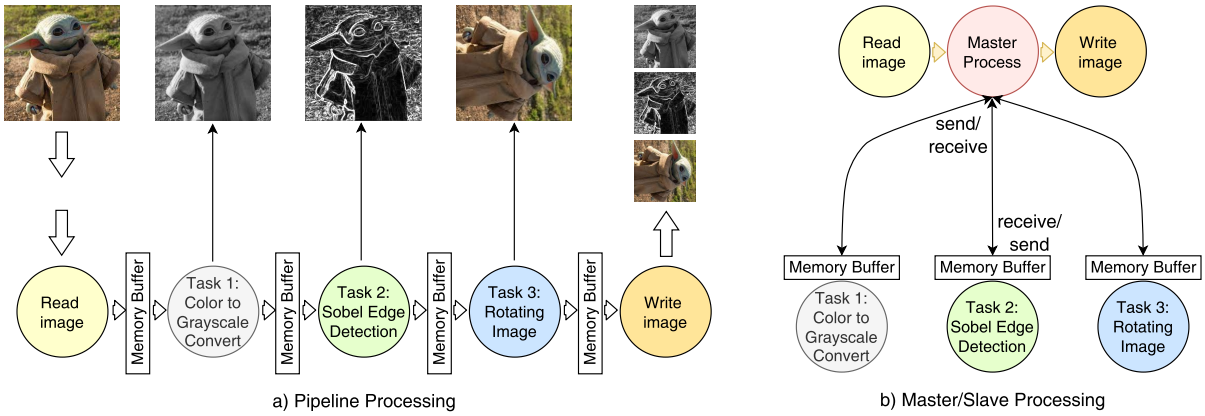


Figura 4.11: Aplicación de procesamiento de imágenes: a) aplicando procesamiento encauzado, b) aplicando procesamiento maestro/esclavo. Fuente: Elaborado por el Autor.

4.6.2 *DESCRIPCIÓN DE LOS ALGORÍTMOS PARA LA APLICACIÓN DE PRUEBA DE SKELETONCORE*

El procesamiento digital de imágenes está presente en aplicaciones industriales, ciencias médicas, biometría e identificación, agricultura, ganadería, satélites de observación terrestre, como en tantas otras tareas de distinta índole en la sociedad. Es por ello que hoy en día se enfatiza mucho en desarrollar nuevas y mejores técnicas para llevar a cabo estos tratamientos en las diversas actividades donde se las requiera.

El análisis y procesamiento de imágenes digitales se realiza a través de algoritmos y sistemas de computación, debido a la complejidad y el número de cálculos necesarios para realizarlo. Es por esto que, si bien la formulación matemática necesaria para su realización data de varias décadas, la posibilidad real de utilizarla de forma cotidiana en la práctica en diferentes áreas ha sido posible recién en la última década, gracias al avance en el hardware de las tecnologías de los dispositivos de computación.

La actual variedad de técnicas, algoritmos y desarrollos de software y hardware utilizados en el procesamiento de imágenes digitales escapa al alcance del presente trabajo. Sin embargo, como aplicaciones para la prueba de concepto de los esqueletos se emplean algunas técnicas de uso común que permiten el análisis de imágenes.

En [Frery and Perciano, 2013] se explican conceptos y nociones básicas al respecto, y en la presente sección se introducen las primeras nociones y conceptos para abordar el procesamiento de imágenes digitales, entre los que se cuentan las formatos de lectura y representación de imágenes, las operaciones de modificación de sobel, las transformaciones de imagen de color a tonalidades de grises, y la generación de efectos sobre regiones de una imagen.

4.6.2.1 La imagen digital y sus características

A) Representación de la forma y color en la imagen: Una imagen digital es la representación de un objeto real, y se pueden representar como mapas de bits o imágenes vectoriales.

Una imagen de dos dimensiones es una función $f(x, y)$ donde x e y representan las coordenadas en el plano X, Y, donde $f(x, y)$ representa la intensidad o nivel de gris o color de la imagen en ese punto. Si x e y son discretos y finitos, entonces la imagen es digital

Como mapas de bits (Bitmaps) las imágenes se representan como una matriz de pixeles. Un Bitmap es un modo elemental para representar imágenes digitales como información en el hardware, específicamente en

la memoria, de un computador. Consiste, básicamente, en formar arreglos de elementos (vectores, matrices, tensores) ordenados de modos específicos.

En general, para el caso de imágenes 2D o en dos dimensiones, se realiza un ordenamiento por filas de elementos de matriz (pixels) asignando a cada uno un valor que determina "el color" en esa posición de la imagen. En el caso de imágenes en tonalidades de grises, el valor del elemento de matriz es un escalar o un sólo canal; mientras que para el caso de imágenes a color el valor de cada elemento de matriz es un vector de tres coordenadas o canales, cada una de las cuales especifica el grado de la combinanción resultante de los colores rojo (Red "R"), verde (Green "G") y azul (Blue "B"). A este modo se le denomina representación RGB. Existen otros modos de representación a color, como por ejemplo CMYK (cián, magenta, amarillo y negro).

Comúnmente, se emplean escalas que determinan rangos dinámicos de 2^N bits, y se denominan $N - bits$. Es decir, para el caso más común de 8-bits, la escala es [0, 255], ya que se define el rango como [0, $2^N - 1$]. La razón del uso común de pixels de 8-bits se basa, principalmente, en que estudios biométricos muestran que el ojo humano no es suficientemente sensible para diferenciar más de 256 niveles de intensidad para un color dado. Además, el rango de valores para los elementos de la matriz de la imagen determinan la capacidad de memoria para almacenarla en el computador.

Es por ello que, para imágenes con tonos de grises, conocidas como "de una banda o canal" el rango de valores de los elementos de la matriz (escalares) es [0, 255], mientras que para imágenes a color, los valores de los elementos de la matriz (vectores de 3 coordenadas) asumen valores en los rangos de ([0, 255],[0, 255],[0, 255]). Esto último significa que todos los colores en el rango visible pueden representarse como combinaciones RGB, variando desde el negro (0,0,0) al blanco (255,255,255). Por lo tanto, una imagen RGB es representada por un arreglo bidimensional de pixeles, cada uno codificado en 3 bytes (3 canales) pudiendo asumir 2.563 diferentes valores de combinaciones, es decir 16.8 millones de diferentes colores, aproximadamente.

B) Resolución, tamaño de imagen y tamaño del archivo de una imagen: La resolución es un concepto presente en todo el proceso de procesamiento de información digital, desde la captura o generación hasta la representación, y afecta (condiciona) el procesamiento posterior.

La resolución, el tamaño de imagen y el tamaño de archivo son tres conceptos están estrechamente relacionados y dependen mutuamente, aunque se refieren a características diferentes. La resolución de una imagen es la cantidad de pixels que la describen. Y una medida típica es en términos de "pixels por pulgada" (ppi). Por tanto, la calidad de la representación así como el tamaño de la imagen dependen de la resolución, que determina a su vez los reqerimientos de memoria para el archivo gráfico a generar.

El tamaño de una imagen son sus dimensiones reales en términos de anchura y altura una vez impresa, mientras que el tamaño del archivo se refiere a la cantidad de memoria física necesaria para almacenar la información de la imagen digitalizada en cualquier soporte de almacenamiento.

La resolución de la imagen condiciona fuertemente estos dos conceptos anteriores, ya que la cantidad de pixels de la imagen digital es fijo y por tanto al aumentar el tamaño de la imagen se reduce la resolución y viceversa. A modo de ejemplo: si se duplica la resolución de una imagen digital, de 50 ppi a 100 ppi, el tamaño de la imagen se reduce a la cuarta parte del original mientras que dividir la resolución por 2. Es decir, se pasa de 300 ppi a 150 ppi obteniendo una imagen con el doble de las dimensiones originales que represebtan cuatro veces su superficie.

Como consecuencia, la reducción de la resolución de la imagen, manteniendo su tamaño, provoca eliminación de pixels. Entonces, se obtiene una representación (descripción) menos precisa de la imagen,

así como transiciones de color más bruscas. El tamaño del archivo que genera una imagen digitalizada es proporcional, como se espera, a la resolución, por lo tanto, variarla implica modificar en el mismo sentido el tamaño del archivo.

4.6.2.2 Algorítmo de Procesamiento de imagen: Conversión de Imagen a Color a Grises

Una imagen es un arreglo matricial de dos dimensiones que aporta información de la intensidad de la luz presente para cada punto de la imagen.

En una imagen a color el valor de cada elemento de matriz es un vector de tres coordenadas o canales, cada una de las cuales especifica el grado de la combinanción resultante de los colores rojo (Red "R"), verde (Green "G") y azul (Blue "B"). El espacio del color RGB es el modelo más conocido y utilizado. El color no es en realidad un atributo que pueda asignársele a los objetos como tal.

En una imagen en escala de grises el valor de cada píxel posee un valor equivalente a una graduación de gris. Las imágenes representadas de este tipo están compuestas de sombras de grises. En el caso de imágenes en tonalidades de grises, el valor del elemento de matriz es un escalar o un sólo canal de 8-bits, la escala es [0, 255], ya que se define el rango como $[0,\ 2^N - 1]$.

La ecuación matemática que expresa este proceso de conversión parte de la ponderación de cada componente de color que indica la sensibilidad del ojo humano a las frecuencias del espectro cercanas al rojo (R), verde (G) y azul (B). Por tanto, para realizar esta conversión se aplica la ecuación 4.7 a cada pixel (x, y) de la imagen true-color. Esto genera una nueva matriz de un byte por pixel que daría la información de luminancia en escala de grises.

$$f(x, y) = R * 0{,}3 + G * 0{,}59 + B * 0{,}11 = nivel\ de\ gris. \tag{4.7}$$

Si quisiéramos convertir a escala de grises una imagen a color, solo hay que convertir cada pixels a color de la paleta a sus pixels de grises correspondientes según la ecuación anterior. Se pueden comparar cualitativamente cada uno de los modelos de representación de las imágenes a color en su equivalente en escala de grises a partir del modelo RGB. La imagen generada se muestra en la Figura 4.12.

Figura 4.12: Ejemplo de imagen a color convertida a escala de grises (imagen original de baby Yoda, en The Mandalorian). Fuente: Elaborado por el autor.

4.6.2.3 Algorítmo de Procesamiento de imagen: Sobel Edge Detection

En el procesamiento de imágenes digitales, un borde o controno es la frontera que delimita o separa dos objetos. Por ello, la detección de bordes (edge detection) es una aplicación común y útil porque permite identificar, medir objetos o segmentar una imagen digital.

Un algoritmo muy usado es el detector de bordes de Sobel. Como se explica en [Trussell and Vrhel, 2008], este algoritmo utiliza un par de máscaras de convolución de 3 x 3, una que estima el gradiente horizontal G_x en la dirección del eje x, y la otra que estima el gradiente vertical G_y en la dirección del eje y. El detector Sobel es muy sensible al ruido en las imágenes, las resalta efectivamente como bordes.

Si definimos A como la imagen original, el resultado de aplicarle las dos imágenes G_x y G_y, que representan para cada punto las aproximaciones horizontal y vertical de las derivadas de intensidades:

$$G_x = \begin{pmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} * A \quad y \quad G_y = \begin{pmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{pmatrix} * A \tag{4.8}$$

Así, en cada punto de la imagen, los resultados de las aproximaciones de los gradientes horizontal y vertical pueden ser combinados para obtener la magnitud del gradiente, mediante:

$$G = \sqrt{(G_x)^2 + (G_y)^2} \tag{4.9}$$

Además, los dos filtros discretos descritos arriba en las ecuaciones 4.8 y 4.9 pueden ser separados así:

$$\begin{aligned} G_x &= \begin{pmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} * A = \begin{pmatrix} +1 \\ +2 \\ +1 \end{pmatrix} * \begin{pmatrix} -1 & 0 & +1 \end{pmatrix} * A \\ G_y &= \begin{pmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{pmatrix} * A = \begin{pmatrix} -1 \\ 0 \\ +1 \end{pmatrix} * \begin{pmatrix} +1 & +2 & +1 \end{pmatrix} * A \end{aligned} \tag{4.10}$$

Entonces, el algoritmo para calcular cada pixel de la Imagen f(x,y) es el mostrado en las siguientes ecuaciones:

a) Magnitud del gradiente horizontal:

$$G_x = f(x-1,y-1)*1+f(x-1,y)*2+f(x-1,y+1)*1\breve{}f(x+1,y-1)*1-f(x+1,y)*2-f(x+1,y+1)*1 \tag{4.11}$$

b) Magnitud del gradiente vertical:

$$G_y = f(x-1,y-1)*1+f(x,y-1)*2+f(x+1,y-1)*1\breve{}f(x-1,y+1)*1-f(x,y+1)*2-f(x+1,y+1)*1 \tag{4.12}$$

c) Magnitud del vector Gradiente:

$$G = \sqrt{(G_x)^2 + (G_y)^2} \tag{4.13}$$

d) Dirección del vector Gradiente:

$$\Theta G = ArcTan(Gy/Gx) \tag{4.14}$$

Como se observa en la Figura 4.13, el operador Sobel calcula (ecuación 4.10) el gradiente de la intensidad de una imagen en cada punto (píxel). Así, para cada punto, este operador da la magnitud del mayor cambio posible, la dirección de este y el sentido desde oscuro a claro. El resultado muestra cómo de forma brusca o suave cambia una imagen en cada punto analizado y, en consecuencia, cuán probable es que este represente un borde en la imagen y, también, la orientación a la que tiende ese borde.

Figura 4.13: Ejemplo de imagen en escala de grises pasada por el filtro Sobel.
Fuente: Elaborado por el autor.

4.6.2.4 Algorítmo de Procesamiento de imagen: Rotating Image

Los cálculos detrás de la acción de rotar una imagen realmente son bastante básicos. En matemáticas, la rotación es un concepto que tiene su origen en la geometría. Cualquier rotación es un movimiento definido en un determinado espacio que conserva al menos un punto en su posición original. Puede describir, por ejemplo, el giro de un cuerpo rígido alrededor de un punto fijo. Una rotación es diferente a otros tipos de movimientos (como la traslación, que no tiene puntos fijos; o la reflexión).

Para un espacio n-dimensional, la rotación se caracteriza por presentar un plano (n-1)-dimensional completo de puntos fijos. Una rotación en el sentido de las agujas del reloj se considera por convenio una magnitud negativa, y de forma análoga, un giro en el sentido contrario a las agujas del reloj tiene una magnitud positiva.

Matemáticamente, una rotación es una aplicación. Todas las rotaciones sobre un punto fijo forman un grupo bajo unas reglas de composición, denominado grupo de rotación (de un espacio en particular). Por ejemplo, en dos dimensiones, girar un cuerpo en el sentido del reloj alrededor de un punto donde se mantienen los ejes fijos, equivale a girar los ejes en sentido contrario a las agujas del reloj alrededor del mismo punto mientras el cuerpo se mantiene fijo. Estos dos tipos de rotación se denominan transformaciones activas y pasivas.

En dos dimensiones, para llevar a cabo una rotación usando una matriz, el punto (x, y) que se gira hacia la izquierda, se escribe como un vector columna y se multiplica por una matriz de rotación calculada a partir del ángulo θ (ecuación 4.15):

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos\ \theta & -sin\ \theta \\ sin\ \theta & cos\ \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{4.15}$$

Las coordenadas del punto (x, y) después de la rotación son (x', y'), y la ecuación 4.16 muestra las fórmulas para obtener x' y y':

$$\begin{aligned} x' &= x * cos\ \theta - y * sin\ \theta \\ y' &= x * sin\ \theta + y * cos\ \theta \end{aligned} \tag{4.16}$$

Donde, los vectores $\begin{bmatrix} x \\ y \end{bmatrix}$ y $\begin{bmatrix} x' \\ y' \end{bmatrix}$ tienen la misma magnitud y están separados por un ángulo θ como se esperaba.

Como ejemplo, en la Figura 4.14 se puede observar una imagen a color rotada 90 grados hacia la derecha.

Figura 4.14: Ejemplo de imagen a color rotada 90 grados en sentido del reloj. Fuente: Elaborado por el autor.

4.7 IMPLEMENTACIÓN Y EVALUACIÓN DEL CÓDIGO SECUENCIAL PARA PROCESAMIENTO DE IMÁGENES

La implementación en C/C++ de la aplicación secuencial de procesamiento de imagenes de prueba (mostrada en el código fuente 4.9) se ejecuta o corre sobre un sólo núcleo del procesador CPU, y sus métricas de tiempo se usan para hacer un análisis compativo de tiempos, rendimiento y aceleración con respecto a los esqueletos paralelos reconfigurables del catálogo de la interfaz de programación paralela **SkeletonCoRe** mostrados en los códigos fuente de **PipeSkeleton** en 5.14, y de **TaskSkeleton** en 6.16.

4.7.1 *DESCRIPCIÓN DEL EXPERIMENTO DE EVALUACIÓN DEL CÓDIGO SECUENCIAL*

En ésta sección se describe la prueba que se realiza sobre el programa secuencial en C/C++ mostrado en el Código Fuente 4.9. Es la misma aplicación de procesamiento digital de imágenes que ya se explico en la anterior **subsección 4.6.6 sobre "Diseño General del Experimento de Evaluación de SkeletonCoRe"**, donde se aplican tres operadores de imagen en forma secuencial, siguiendo el orden: Imagen de Entrada → Tarea 1: Conversión de imagen en Color RGB a imagen en Escala de Grises (RGB Color to Grayscale Converter) → Tarea 2: Operador Sobel para Detección de Bordes (Sobel's Edge Detection) → Tarea 3: Operador de Interpolación (Resizing Interpolation) → Imagen de Salida.

La ejecución de éste código secuencial se realiza sobre un sólo núcleo del CPU, y con un solo hilo de ejecución, en una configuración similar a la mostrada en la Figura 4.16. Este programa secuencial implementado en C/C++ es usado modelo de evaluación de referencia para comparar los tiempos y costos de ejecución secuencial (sin usar esqueletos) con respecto a la ejecución paralela de los kernels de este mismo programa usando los esqueletos algoritmicos reconfigurables desarrollados en la presente tesis en lenguage OpenCL C/C++.

4.7.2 *PROGRAMACIÓN DEL CÓDIGO SECUENCIAL EN C/C++ PARA PROCESAMIENTO DE IMÁGENES*

En esta sección se muestra el código fuente secuencial de prueba en C/C++ en extenso, donde se puede ver la solución para el procesamiento digital de imágenes. Así, en el siguiente código fuente 4.9 se observa la implementación en C/C++ puro, sin usar esqueletos ni OpenCL, de la aplicación de procesamiento digital de imágenes.

```
//*************************************************************************************************
//* Nombre del programa:  app-seq-dip-7-final.cpp
//* Programador o autor:  Carlos Acosta-León
//* Función del programa: Programa secuencial que toma imágenes en formato .jpeg
//*                       (en color o en escala de grises) con dimensiones 256x256, 512x512,
//*                       1024x1024 y 2048x2048 pixels y les aplica operadores para procesamiento
//*                       de imágenes digitales como Conversión de Color a Grises, Sobel Edge
//*                       Detection e Interpolación por "Rotate or Resize images".
//* Lenguaje de program:  Lenguaje C/C++ (en Linux Ubuntu LTS 20.04)
//* Fecha:                25 de Febrero de 2023
//* Motivo:               Programa fuente usado como ejemplo en Tesis Doctoral Ciens de la Comp
//*************************************************************************************************
#include "app-seq-proof-dip-api-7-final.hpp"

// Declaración de espacios de nombres de las clases para Proc. Digital de Imágenes
using namespace imagenes;

/*****************************************************/
/*** PROGRAMA PRINCIPAL                            ***/
/*---------------------------------------------------*/
int main(int argc, char *argv[]) {
    /* Declarando variables para la imagen de entrada */
    int            Nimages;        // Cantidad imágenes de entrada a procesar
    int            RGBchannels;    // RGB: 3 channels if color image
    int            Graychannel;    // One (1) channel if grayscale image
    float          t_exec_task1, total_exec_task1; // Variables para medir el tiempo de procesamiento
    float          t_exec_task2, total_exec_task2; // Variables para medir el tiempo de procesamiento
    float          t_exec_task3, total_exec_task3; // Variables para medir el tiempo de procesamiento
    float          t_total_exec;   // Variables para medir el tiempo de procesamiento
    float          t_write, t_total_write;// Variables para medir el tiempo de procesamiento
    float          t_read, t_total_read;// Variables para medir el tiempo de procesamiento

    /* Verificando cantidad de parametros de entrada */
    if (argc < 2) {
        printf("Use: ./exec-program image_input.jpg Num(CatidadImagenes)\n");
        return 1;
    }

    // Lee numero de imágenes a procesar
    Nimages = 256; //(int)(*argv[2]);

    //Declarar aqui el objeto imagen
    imagen      inImage;        // Crea el objeto imagen de entrada
    imageKernel operadorImagen; // Crea el objeto con las operaciones de imagen

    /*********************************************************************/
    // Reserva espacio de memoria para las imagenes de entrada y salidas
    inImage.ent_Imagen = (unsigned char*)malloc(inImage.width * inImage.height *
        inImage.RGBChannels * sizeof(inImage.ent_Imagen));
```

Código fuente 4.9: Ejemplo del código fuente secuencial en C/C++ donde se muestra el programa principal de la aplicación para el procesamiento de imágenes que aplica la cadena de operadores de imagen (sin esqueletos ni OpenCL). Remítase al anexo B.17 para acceder a las definiciones de las funciones y procedimiento usados en éste. Fuente: Elaborado por el autor.

```
50
51      /*********************************************************************/
52      /* Lee y descomprime la imagen de entrada .jpeg (grayscale or color) and parameters (metadata) */
53      cout << "-----------------------------------------------------" << "\n";
54      //cout << "---> Leyendo imágenes de entrada!" << "\n";
55
56      //* INICIO DEL TIEMPO TOTAL
57      t_total_exec = clock();
58    for (int i = 0; i < Nimages; i++) { // Begin for
59      /* Lee y descomprime la imagen de entrada .jpeg (grayscale or color) and parameters (metadata) */
60      /* Se mide el tiempo de lectura de datos de entrada */
61      t_read = clock();
62          inImage.readInputImageJPG(argv[1], &inImage.ent_Imagen);
63      t_total_read = (clock() - t_read) + t_total_read;
64
65      /*********************************************************************/
66      //cout << "---> Inicia el procesamiento!" << "\n";
67      /* Tarea 1: Convert from color .jpeg image to gray image for sobel edge detection processing */
68      // Se ejecuta la operacion de imagen y captura el resultado
69      t_exec_task1 = clock();
70          operadorImagen.convertImageGrayscale(inImage.ent_Imagen, &inImage.res_imageGrayscale,
71              inImage.width, inImage.height, &inImage.GrayChannel);
72      total_exec_task1 = (clock() - t_exec_task1) + total_exec_task1;
73
74      /*********************************************************************/
75      /* Tarea 2: Applying sobel edge detection processing for grayscale image */
76      // Se ejecuta la operacion de imagen y captura el resultado
77      t_exec_task2 = clock();
78          operadorImagen.sobelEdgeDetection(inImage.res_imageGrayscale, &inImage.res_imageSobel,
79              inImage.width, inImage.height, inImage.GrayChannel);
80      total_exec_task2 = (clock() - t_exec_task2) + total_exec_task2;
81
82      /*********************************************************************/
83      /* Tarea 3: Applying another image operator as image Interpolation */
84      // Se ejecuta la operacion de imagen y captura el resultado
85      t_exec_task3 = clock();
86          operadorImagen.imageRotate_(inImage.ent_Imagen, &inImage.res_imageRotated, inImage.width,
87              inImage.height, inImage.RGBChannels);
88      total_exec_task3 = (clock() - t_exec_task3) + total_exec_task3;
89
90      /*********************************************************************/
91      //* Grabando y comprimiento en jpg las imágenes de entrada y las resultados
92      //* Graba en archivo imagen original de entrada a color o a escala de grises
93      t_write = clock();
94          inImage.writeOutputImageJPG("image-original-out-0.jpg", inImage.width, inImage.height,
95              inImage.RGBChannels, inImage.ent_Imagen, 95);
96          //* Graba en archivo la imagen de salida a Escala de 255 Grises
97          inImage.writeOutputImageJPG("image-grayscale-out-1.jpg", inImage.width, inImage.height,
98              inImage.GrayChannel, inImage.res_imageGrayscale, 95);
99          //* Graba en archivo de salida imagen con sobel edge detection
100         inImage.writeOutputImageJPG("imagen-sobel-out-2.jpg", inImage.width, inImage.height,
101             inImage.GrayChannel, inImage.res_imageSobel, 95);
102         //* Graba en archivo de salida imagen con imagen rotada 'N' grados
103         inImage.writeOutputImageJPG("image-interpol-out-3.jpg", inImage.width, inImage.height,
104             inImage.RGBChannels, inImage.res_imageRotated, 95);
105     t_total_write = (clock() - t_write) + t_total_write;
106   }// End for
107     //* FIN Y SUMA DEL TIEMPO TOTAL
108     t_total_exec = (clock() - t_total_exec);
109
110     /*********************************************************************/
```

(Cont. Codigo fuente 4.9) Ejemplo del código fuente secuencial en C/C++ donde se muestra el programa principal de la aplicación para el procesamiento de imágenes que aplica la cadena de operadores de imagen (sin esqueletos ni OpenCL). Remítase a la sección de anexos para acceder a las definiciones de las funciones y procedimiento usados en éste. Fuente: Elaborado por el autor.

```
111     /* Se miden los tiempos de procesamiento: Tiempo usado de CPU = fin - inicio */
112     cout << "Input Files                    - Reading time is     : " << setprecision(6) <<
113     (t_total_read/(CLOCKS_PER_SEC)*MILLISEC)
114         << " milliseconds\n";
115     /* Se mide el tiempo total de procesamiento de TAREA 1 */
116     cout << "Task 1: Converting to Grayscale - The elapsed CPU time: " << setprecision(4)
117         << (total_exec_task1/(CLOCKS_PER_SEC)*MILLISEC) << " milliseconds\n";
118     /* Se mide el tiempo total de procesamiento de TAREA 2 */
119     cout << "Task 2: Sobel Edge Detection    - The elapsed CPU time: " << setprecision(4)
120         << (total_exec_task2/(CLOCKS_PER_SEC)*MILLISEC) << " milliseconds\n";
121     /* Se mide el tiempo total de procesamiento de TAREA 3 */
122     cout << "Task 3: Rotating Image          - The elapsed CPU time: " << setprecision(4)
123         << (total_exec_task3/(CLOCKS_PER_SEC)*MILLISEC) << " milliseconds\n";
124     cout << "Output Files                    - Writing time is     : " << setprecision(4)
125         << (t_total_write/(CLOCKS_PER_SEC)*MILLISEC) << " milliseconds\n";
126     /* Se mide el tiempo total de procesamiento */
127     cout << "Tiempo Total de Proc.           - The ELAPSED CPU TIME: " << setprecision(4)
128         << (t_total_exec/(CLOCKS_PER_SEC)*MILLISEC) << " milliseconds\n";
129     //cout << "---> Finaliza el procesamiento!" << "\n";
130
131     /*************************************************************************/
132     // Liberando espacio de memoria asigado a arreglos de imágenes
133     //free(imageColor);
134     delete(inImage.ent_Imagen);
135     delete(inImage.res_imageGrayscale);
136     delete(inImage.res_imageSobel);
137     delete(inImage.res_imageRotated);
138
139     /* Terminación exitosa! */
140     cout << "---> Finished processing, successful completion!" << "\n";
141     cout << "-----------------------------------------------------" << "\n";
142     return 0;
143 } /*** FIN PROGRAMA PRINCIPAL ***/
```

(Cont. Codigo fuente 4.9) Ejemplo del código fuente secuencial en C/C++ donde se muestra el programa principal de la aplicación para el procesamiento de imágenes que aplica la cadena de operadores de imagen (sin esqueletos ni OpenCL). Remítase a la sección de anexos para acceder la cabecera o "header" (.hpp) con las definiciones de variables, funciones y procedimiento usados en éste. Fuente: Elaborado por el autor.

4.7.3 *EVALUACIÓN DEL CÓDIGO SECUENCIAL EN C/C++ PARA PROCESAMIENTO DE IMÁGENES*

A continuación se muestra el comando de compilación y la corrida del programa en C/C++ con los operadores de imagenes usados para el procesamiento secuencial de imágenes (Ver el código fuente 4.9). El código implementado en lenguaje C/C++ puro, corre secuencialmente tres operadores de procesamiento de imagen como "Color to Grayscale image Converter", "Sobel's Edge Detection" y "Rotating Image". Además, este programa se corre usando un solo núcleo del CPU, y sin ningun tipo de esqueleto algoritmico.

En la siguiente traza de salida (consola de comandos de Ubuntu Linux) se imprimen en pantalla los resultados cuantitativos del procesamiento de 256 imagenes de 1024 x 1024 pixeles, así obtenemos la siguiente salida por consola:

```
$> g++ app-seq-dip.cpp -std=gnu++2a -O3 -o app-seq-dip.out -ljpeg -fpermissive
$> ./app-seq-dip.out imagen-entrada-1024x1024.jpg
Imput Image dimensions: Width = 1024 pixels, Height = 1024 pixels, Channels = 3
Input Files                    - Reading time is     : 1795.97 milliseconds
Task 1: Converting to Grayscale - The elapsed CPU time: 586.612 milliseconds
Task 2: Sobel Edge Detection    - The elapsed CPU time: 3707.18 milliseconds
Task 3: Rotating Image          - The elapsed CPU time: 1118.27 milliseconds
Output Files                   - Writing time is     : 4606.36 milliseconds
Total time of Proc.            - The ELAPSED CPU TIME: 11815.1 milliseconds
```

```
Total time of I/O              - The ELAPSED I/O TIME: 6402.33 milliseconds
Images processed: 256
---> Finished processing, successful completion!
```

Las imágenes resultantes del procesamiento se pueden observar en la Figura 4.15, donde se separa el resultado por cada operador de imagen respectivo. Se evidencia que las imágenes procesadas son las esperadas según cada algoritmo de procesamiento de imagen aplicado. Es decir, en la Figura 4.15 a) se tiene la imagen de entrada original a color RGB (formato .jpg), luego en b) se obtiene a partir de la imagen original a color una imagen convertida a escala de 256 niveles de grises; en c) se produjo una imagen donde se resaltan los bordes de los objetos detectados (con máscara Sobel), y en d) se ve la imagen original a color rotada 90 grados hacia la derecha.

Figura 4.15: Imagenes resultantes del procesamiento digital: a) Imagen original a color RGB (formato .jpg), b) Imagen convertida de color a escala de 256 grises, c) Imagen con bordes de objetos resaltados (con máscara Sobel), y d) Imagen con Interpolación. Fuente: Elaborado por el autor.

Además de los resultados cualitativos obtenidos en la Figura 4.15, en la Tabla 4.2 se presentan los resultados cuantitativos a través de sus gráficos adjuntos. En estos se presentan los resultados y curvas de variación de los tiempos secuenciales de procesamiento de cuatro listas de 256 imagenes c/u, una con imágenes de dimensiones 256x256, otra con 512x512, la siguiente con 1024x1024 y la última con 2048x2048 pixeles.

En este experimento de evaluación se observa que los tiempos de procesamiento secuencial se incrementan cuando se aumenta el tamaño de los datos o imágenes de entrada. Es decir, hay un aumento significativo del tiempo de procesamiento en CPU en la medida que se procesan las listas de imágenes de 256x256 pixeles (con menor cantidad de datos), pasando por las listas de 512x512 y 1024x1024 pixeles, hasta las imágenes de 2048x2048 pixeles (mayor cantidad de datos), notando una diferencia de hasta dos órdenes de magnitud

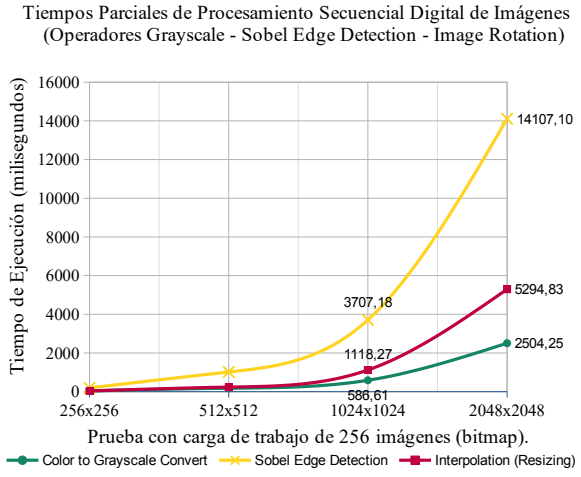
(aprox. 100 veces) del tiempo de CPU entre 256 piexeles (con 196,07 milisegundos) y 2048 piexeles (con 14107,10 milisegundos) en el caso de operador Sobel's Edge detection, lo cual también se observó en los demás operadores de imagen. Esto se mantiene a pesar de que la diferencia del tiempo de procesamiento de cada operador es notable, siendo el operador Sobel' Edge Detection el que consume sustancialmente mayor tiempo del procesador o CPU.

Estos tiempos son tomados como referencia para la comparación y determinación del índice de aceleración del cómputo den los esqueletos PipeSkeleton y TaskSkeleton configurados sólo en GPU (0-3-0), sólo en FPGA (0-0-3) y las combinaciones de particiones hardware/software entre CPU, GPU y FPGA (1-1-1).

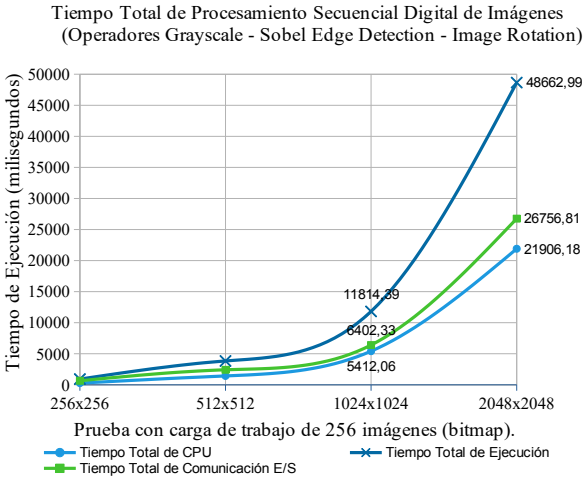
Tabla 4.2: Métricas de tiempos de ejecución de la aplicacion secuencial en C/C++ para proc. de imágenes.

CÓDIGO SECUENCIAL CON C++ PURO, SIN USO DE ESQUELETOS ALGORÍTMICOS										
Configuración	Carga de trabajo (Workload)		Tiempo de Color to Grayscale Convert	Tiempo de Sobel Edge detection	Tiempo de Image Rotation	Tiempo Total de CPU	Tiempo lectura archivos (Entrada)	Tiempo escritura archivos (Salida)	Tiempo Total de Comunicación E/S	Tiempo Total de Ejecución
Partición (CPU-GPU-FPGA)	Cantidad de Imagenes (Carga de Trabajo)	Dimensión NxN o Tamaño de Imagen (pixels)	Tiempo proc. Color to Grayscale x 256 imágenes (miliseg)	Tiempo proc. Sobel Edge Detection x 256 imágenes (miliseg)	Tiempo proc. Image Rotation x 256 imágenes (miliseg)	Tiempo CPU x 256 imágenes (Sin Esqueleto x 1 núcleo) (milisegs.)	Tiempo de lectura x 256 imágenes (miliseg.)	Tiempo de grabar x 256 imágenes (milisegs.)	Tiempo Total Comunicación x 256 imágenes (miliseg)	Tiempo CPU + Tiempo Comunic. (miliseg)
Caso Secuencial en C/C++, CPU, sin OpenCL	256	256x256	42,17	196,07	33,90	272,14	194,94	449,89	644,83	916,98
		512x512	176,46	1015,50	234,15	1426,10	712,91	1701,22	2414,13	3840,23
		1024x1024	586,61	3707,18	1118,27	5412,06	1795,97	4606,36	6402,33	11814,39
		2048x2048	2504,25	14107,10	5294,83	21906,18	6563,91	20192,90	26756,81	48662,99

a)



b)



c)

Leyenda: Se muestras los gráficos b) y c) asociados a la tabla a) con los resultados de las mediciones de los tiempos de ejecución secuencial de la aplicación para procesamiento de imágenes sobre un sólo núcleo del CPU. En esta aplicación se usaron los operadores de imagen: Color2Grayscale Convert, Sobel's Edge Detection e Image Resize by Interpolation. se procesaron 256 imágenes cuadradas (NxN) en cadena, con dimensiones 256, 512, 1024 y 2048 pixels, si usar OpenCL ni Esqueletos. Fuente: Elaborado por el autor.

4.8 Implementación y Evaluación del Esqueleto Secuencial SEQSkeleton

En la programación secuencial se acostumbra a dividir el programa en módulos que representan subrutinas, procedimientos o funciones los cuales agrupan o encapsulan tareas secuenciales. Por esto, en la PAPI **SkeletonCoRe** se incluye un esqueleto secuencial el cual es un modelo de computación trivial de uso común por los prgramadores. A este esqueleto se le ha denominado **SEQSkeleton** y permite al programador agrupar

una o mas tareas secuenciales encapsuladas, como tareas (Kernels) de cómputo intensivo preferiblemente, sin considerar alguna implementación en OpenCL, sino usando solo construcciones nativas del lenguaje C/C++.

4.8.1 *ESPECIFICACIÓN DEL ESQUELETO TRIVIAL SECUENCIAL SEQSKELETON*

Este esqueleto se ha implementado como un patrón de computación *"secuencial"* de instrucciones el cual procesa unos datos entrada mediante una serie finita de n instrucciones o tareas, produciendo unos resultados de salida. Este patrón puede representarse así, f: $\alpha \rightarrow \beta$, donde α representa el estado inicial del algoritmo, $\alpha \rightarrow I_0, I_1, \ldots, I_i, \ldots, I_{n-1} \rightarrow \beta$, con β su estado final.

Al igual que los demás esqueletos de la Interfaz de Programación de Aplicaciones Paralelas (SkeletonCoRe), el esqueleto *SEQSkeleton* se diseña como una plantilla de alto nivel que se instancia con varios parámetros, entre datos y comportamientos **(ver código fuente ??)**. Estos parámetros son: la partición de software, un arreglo de tareas o kernels ordenados secuencialmente, los datos y sus dimensiones.

Un ejemplo del uso y parámetros del esqueleto se muestra a continuación:

SEQSkeleton*(CPUpartition, seqTasksList[], in_file, out_file)*,

donde: *CPUpartition* es la estructura de datos que contiene toda la información de configuración para el procesamiento secuencial, *seqTasksList* es el arreglo de apuntadores a las tareas ordenadas serialmante que conforman el programa secuencial, y *in_file y out_file* son los archivos desde donde se lee y escribe, respectivamente, los datos que se van a procesar y los resultados.

4.8.2 *DESCRIPCIÓN DEL EXPERIMENTO DE EVALUACIÓN DEL ESQUELETO SEQSKELETON*

Como una prueba inicial de concepto de la PAPI **SkeletonCoRe** se procede primero a implementar y evaluar la funcionalidad y rendimiento del esqueleto secuencial *SEQSkeleton*. Es decir, con *SEQSkeleton* se encapsula el código fuente secuencial mostrado en extenso en 4.9, donde se aplican los operadores o kernels de imagen elegidos como aplicación de prueba, secuencialmente a una carga cutaro listas de 256 imágenes a color en formato jpg.

Esta aplicación de prueba aplica varios algorítmos de procesamiento de imágenes como conversión de imagen a color a escala de grises, detección de bordes de objetos en imagen y rotación de imagen en 90 grados. Además, se usa para evaluar la funcionalidad, abstracción y rendimiento de todos los Esqueletos Algorítmicos Reconfigurables de la PAPI **SkeletonCoRe**.

La carga de trabajo para procesamiento está compuesta de cuatro listas de 256 imágenes con diferentes dimensiones: 256x256, 512x512, 1024x1024 y 2048x2048 pixeles, respectivamente, para el procesamiento intensivo.

En la anterior Figura 4.11 se puede ver el diagrama lógico de la aplicación de procesamiento de imágenes para la prueba de concepto de todos los Esqueletos Algorítmicos Reconfigurables. Además, en la siguiente Figura 4.16 se puede ver la configuración usada sólo para realizar la prueba de rendimiento y funcionalidad del esqueleto *SEQSkeleton*.

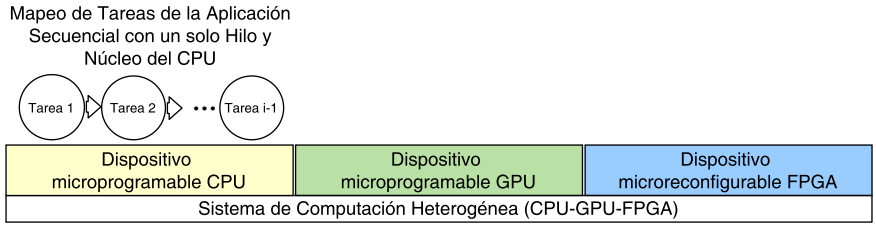


Figura 4.16: Configuración de SEQSkeleton usando un solo hilo sobre un núcleo del microprocesador o CPU del Host. Fuente: Elaborado por el Autor.

Por último, ésta aplicación de procesamiento de imágenes será la usada para la implementación, medición y comparación de las métricas de rendimiento entre los esqueletos paralelos y el secuencial.

4.8.3 *IMPLEMENTACIÓN DEL ESQUELETO TRIVIAL SECUENCIAL SEQSKELETON*

En esta sección, en el código fuente 4.10 se muestra la implementación en C/C++ del esqueleto **SEQSkeleton**. Con este código se compila y corre la aplicación para procesamiento de imagenes de prueba en la cual, como ya se ha explicado, se aplican secuencialmente una cadena de tres operadores de imagen como: *entrada* $\rightarrow$ etapa 1 (Color to Grayscale Convert) $\rightarrow$ etapa 2 (Sobel's Edge Detection) $\rightarrow$ etapa 3 (Rotating Image) $\rightarrow$ *salida*.

Este esqueleto se ejecuta secuencialmente usando un solo hilo o proceso (thread) sobre un solo núcleo del procesador o CPU. Además, tiene como objetivo permitirle al programador la implementación del modelo de computación secuencial, lo cual le obliga a asignar las tareas siempre serialmente, comenzando desde la primera tarea hasta la última tarea serial del programa. Internamente el esqueleto gestiona la creación del hilo para el procesamiento secuencial y el orden secuencial de las tareas dentro de éste hilo, pero sin usar instrucciones del API OpenCL, sino usando solamente instrucciones nativas del lenguaje de programación C/C++.

A modo de comparación, en el anterior código fuente 4.9 se observa una implementación de la aplicación secuencial de prueba sin esqueletos, es decir, programada de forma explícita con instrucciones en lenguaje C/C++ puro (sin la API OpenCL ni esqueletos de la PAPI **SkeletonCore**). Mientras que en el código fuente 4.10 se muestra la plantilla **SEQSkeleton** del catálogo de esqueletos reconfigurables de **SkeletonCoRe** usada para codificar, compilar, correr y probar la misma aplicación secuencial de prueba.

```
//*********************************************************************************************
//* Program name: SEQSkeleton of PAPI SkeletonCORE
//* Programmer  : Carlos Acosta-León
//* Function    : SEQSkeleton Body.
//* Language/API: C/C++
//*********************************************************************************************
#include "user_define_tasks_body_ver_1-4.hpp"   // Always must be the first header file
#include "papi-skeletoncore-header_ver_1-4.hpp"
//*** Definition of Tasks Function +++//
void Task1_convertImageGrayscale(unsigned char *inImage, unsigned char *outImage[], int width, int height,
              int *channels);
void Task2_sobelEdgeDetection(unsigned char *inImage, unsigned char *(outImage[]), int iWidth, int iHeight,
              int channels);
void Task3_imageRotated(unsigned char *inImage, unsigned char *(outImage[]), int width, int height,
              int channels);
//*** Define an Array with pointer to functions: Pointer Array to Processing Functions
void (*tasksArray[MAXSIZE_TASKSLIST])(unsigned char *inImage, unsigned char *outImage[], int width,
    int height, int *channels, double &exec_time) = {
      *Task1_convertImageGrayscale(),
      *Task2_sobelEdgeDetection(),
      *Task3_imageRotated(),
  };
//*** SEQSkeleton Body ***//
void seqSkeleton(skeletoncore::devicePartition CPUpartition, void (*tasksArray[])(void), int num_tasks) {
  // Sequential Processing Skeleton
  cout << "     -> 5. Executing Secuential Generic Skeleton..." << "\n";
    init_();
    for(int i=0; i < num_tasks; i++)
      cout << " Processing Task" << i << "\n"
      (tasksArray[i])(CPUpartition.dataSet.images.input_Images,
         &CPUpartition.dataSet.images.out_Image,
         CPUpartition.dataSet.images.properties.width,
         CPUpartition.dataSet.images.properties.height,
         &CPUpartition.mydataSet.images.properties.channelsGray);
    cout << "=======> End Sequential Processing... 256 images processed!" << "\n";
    finish_();
};
```

Código fuente 4.10: Código con la implementación en C/C++ del Esqueleto **SEQSkeleton**. El detalle del código se puede encontrar en el Anexo C, código fuente B.19. Fuente: Elaborado por el autor.

Ahora, en el código fuente mostrado en 4.11 se puede ver un ejemplo donde se muestra un Programa Principal que usa el Esqueleto Reconfigurable SEQSkeleton y demás componentes de la **PAPI SkeletonCoRe**.

```
//*********************************************************************************************
//* Program name: Library of Objects and Parallel Skeletons of PAPI SkeletonCORE
//* Programmer  : Carlos A. Acosta-León
//* Function    : Show the SEQSkeleton Usage.
//* Language/API: OpenCL C/C++
//* Date        : 25/02/2022
//*********************************************************************************************
#include "papi-skeletoncore-header_ver_1-4.hpp"
//* Define User's Tasks or Functions for processing
#include "user_define_tasks_body_ver_1-4.hpp"
#define BILLION 1000000000L;
const   int MAXSIZE_TASKSLIST = 7;

// Declare space of names for SkeletonCoRe
using namespace skeletoncore_papi;
//using namespace user_tasks_body;
using namespace std;

/******* BEGIN MAIN PROGRAM *********************************************************/
int main(int argc, char *argv[]) {
    //* Declare data variables
    string  imageType     = "jpg"; // type of jpg, png, etc, for example
    string  mytasksPool[3];
    //* FIRST STEP: DECLARE INSTANCES OF SKELETONCORE OBJECTS
    //* Declare skeletoncore objects
    skeletoncore::papi_skeletoncore skelcore;  // Create the setup enviorement
    skeletoncore::dataSet           mydataSet; // Create container for images data/operations
    // device Partition for Sequential Processing on CPU
    skeletoncore::devicePartition   myCPUPartition;
    //* SECOND STEP: SETUP THE SKELETONCORE ENVIORENMENT AND COMPUTING PARTITIONS
    //* Initialize the skeletoncore processing enviorement
    skelcore.init(argc);
    //* Getting info of computing devices in platform
    skelcore.platformDiscovery(skelcore.deviceGroup);
    //* Getting input data: Here it gests all properties from input image readed
    mydataSet.images.readDataSetImage("jpg", argv[1], mydataSet, &mydataSet.images.input_Images,
      timereal_input, timecpu_input);
    //* Definition of Tasks body: Must be inside the "user's_define_tasks_body.hpp" header file
    skelcore.getTasksKernel(mytasksPool);
    //* THIRD STEP: SETUP THE COMPUTING PARTITIONS
    //* Setting Up Partitions for compute devices
    skelcore.partitionSetup(skelcore.deviceGroup, myCPUPartition, mydataSet.images.cpu, mytasksPool[0]);
  //* FOURTH STEP: EXECUTE SEQUENTIAL SKELETON FROM SKELETONCORE PAPI
    skelcore.exec.seqSkeleton(myCPUPartition);
    //* FIFTH STEP: GET AND WRITE RESULTS AFTER PROCESSING
    //* Putting out results into Files
    mydataSet.images.writeOutputImages("image-out-list.jpg", mydataSet.images.properties.width,
      mydataSet.images.properties.height, mydataSet.images.properties.channels, mydataSet.images.out_);
    //* SIXTH STEP: CLOSING SKELETONCORE ENVIORENMENT
    skelcore.terminate();
    return 0; /*** END MAIN PROGRAM ***/
}
```

Código fuente 4.11: Código que muestra un Programa Principal usando el Esqueleto SEQSkeleton de la **PAPI SkeletonCoRe** para procesamiento secuencial. El código fuente de la cabecera (header "papi-skeletoncore.hpp") está en el Anexo C, código fuente B.19. Fuente: Elaborado por el autor.

4.8.4 *EVALUACIÓN DE RESULTADOS DEL ESQUELETO TRIVIAL SECUENCIAL SEQSKELETON*

A continuación se muestran los comandos de compilación y corrida del programa en C/C++ que implementa los operadores de imagenes de prueba usados para el procesamiento secuencial de imágenes, mostrado en el anterior código fuente 4.9.

Como resultado cualitativo, en la Figura 4.17 se observan las imágenes que se generan como resultado de la aplicación de cada tarea con su operador de imagen respectivo de la aplicación de procesamiento de imágenes. A continuación se registra la traza de salida por consola del procesamiento de sólo una lista de 256 imágenes de 1024 x 1024 pixeles, así se obtiene la siguiente salida:

```
$> g++ app-skeletoncore-seqskeleton.cpp -std=gnu++2a -O3 -o app-skeletoncore-seqskeleton.out -ljpeg -fpermissive
$> ./app-skeletoncore-seqskeleton.out imagen-entrada-1024x1024.jpg 256
Imput Image dimensions: Width = 1024 pixels, Height = 1024 pixels, Channels = 3
Input Files                    - Reading time is     : 1795.97 milliseconds
Task 1: Converting to Grayscale - The elapsed CPU time: 586.612 milliseconds
Task 2: Sobel Edge Detection    - The elapsed CPU time: 3707.18 milliseconds
Task 3: Rotating Image          - The elapsed CPU time: 1118.27 milliseconds
Output Files                   - Writing time is     : 4606.36 milliseconds
Total time of Proc.            - The ELAPSED CPU TIME: 11815.1 milliseconds
Total time of I/O              - The ELAPSED I/O TIME: 6402.33 milliseconds
Images processed: 256
---> Finished processing, successful completion!
```

a) Imagen de entrada original (1024x1024). b) Imagen original Convertida a Escala de Grises (1024x1024). c) Imagen con Detección de Bordes con Sobel (1024x1024). d) Imagen original rotada 90 grados (1024x1024).

Figura 4.17: Imagenes resultantes del procesamiento usando el esqueleto **SEQSkeleton**: a) Imagen original a color RGB (formato .jpg), b) Imagen convertida de color a escala de 256 grises, c) Imagen con bordes de objetos resaltados (con máscara Sobel), y d) Imagen con Rotación. Fuente: Elaborado por el autor.

En la Tabla 4.3 y sus gráficos adjuntos se presentan los resultados cuantitativos y curvas de los tiempos secuenciales de procesamiento de cuatro listas de 256 imágenes c/u, una con imágenes de dimensiones 256x256, otra con 512x512, la siguiente con 1024x1024 y la última con 2048x2048 pixeles. Estos resultados cuantitativos muestran, al igual que el caso de la Tabla 4.3 subsección 4.7.3 del código secuencial en C/C++ sin esqueletos, una variación de los tiempos secuenciales de procesamiento de las cuatro listas de 256 imagenes c/u.

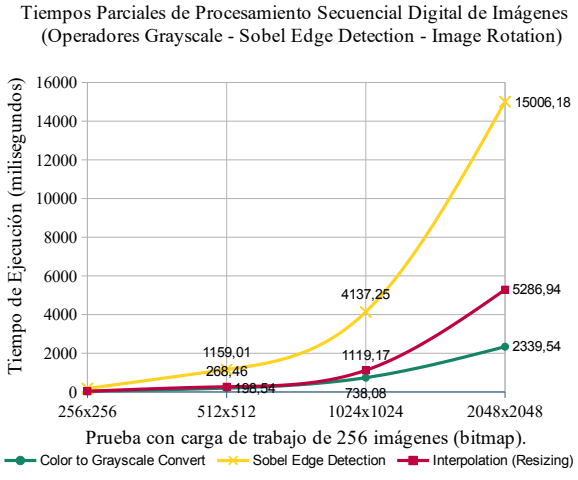
Comparando las medidas de tiempo y sus gráficas de la Tabla 4.2 con la presente Tabla 4.3 se registra un leve incremento del tiempo secuncial de CPU cuando se usó el esqueleto **SEQSkeleton**.

La Tabla 4.2 muestra que los tiempos de procesamiento secuencial también se incrementan cuando se aumenta el tamaño de los datos o imágenes de entrada, dándose un aumento significativo del tiempo de procesamiento en CPU en la medida que se procesan las listas de imágenes de 256x256 pixeles (con menor cantidad de datos), pasando por las listas de 512x512 y 1024x1024 pixeles, hasta las imágenes de 2048x2048 pixeles (mayor cantidad de datos). Se ha notado una diferencia de al menos dos órdenes (alrededor de 100 veces) de magnitud del tiempo de CPU entre 256 piexeles (con 179,10 milisegundos) y 2048 piexeles (con 15006,18 milisegundos) en el caso de operador Sobel's Edge detection, y en los demás operadores de imagen. El operador Sobel' Edge Detection sigue siendo el operador que consume mayor tiempo del procesador o CPU.

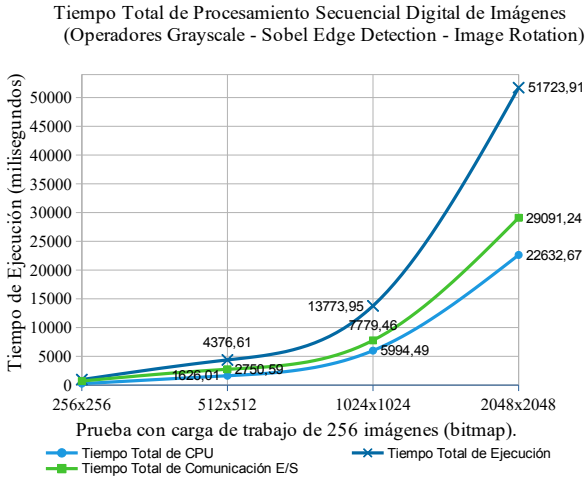
Tabla 4.3: Métricas de tiempos de ejecución de la aplicacion de proc. de imágenes con SEQSkeleton.

CON C++, USANDO EL ESQUELETO SEQSKELETON, PERO SIN OPENCL										
Configuración	Carga de trabajo (Workload)		Tiempo de Color to Grayscale Convert	Tiempo de Sobel Edge detection	Tiempo de Image Rotation	Tiempo Total de CPU	Tiempo lectura archivos (Entrada)	Tiempo escritura archivos (Salida)	Tiempo Total de Comunicación E/S	Tiempo Total de Ejecución
Partición (CPU-GPU-FPGA)	Cantidad de Imagenes (Carga de Trabajo)	Dimensión NxN o Tamaño de Imagen (pixels)	Tiempo proc. Color to Grayscale x 256 imágenes (miliseg)	Tiempo proc. Sobel Edge Detection x 256 imágenes (miliseg)	Tiempo proc. Image Rotation x 256 imágenes (miliseg)	Tiempo CPU x 256 imágenes (Sin Esqueleto x 1 núcleo) (milisegs.)	Tiempo de lectura x 256 imágenes (miliseg.)	Tiempo de grabar x 256 imágenes (milisegs.)	Tiempo Total Comunicación x 256 imágenes (miliseg)	Tiempo CPU + Tiempo Comunic. (miliseg)
Caso Secuencial en C/C++, CPU, sin OpenCL	256	256x256	38,53	179,10	33,05	250,68	197,40	508,12	705,52	956,20
		512x512	198,54	1159,01	268,46	1626,01	835,19	1915,41	2750,59	4376,61
		1024x1024	738,08	4137,25	1119,17	5994,49	2631,02	5148,43	7779,46	13773,95
		2048x2048	2339,54	15006,18	5286,94	22632,67	7365,16	21726,08	29091,24	51723,91

a)



b)



c)

Leyenda: En la Tabla a) se presentan los tiempos de ejecución secuencial de la aplicación de prueba sobre un sólo núcleo del CPU, en b) y c) se ven las gráficas. Se procesaron 256 en cadena, con dimensiones 256, 512, 1024 y 2048 pixels, usando SEQSkeleton en C/C++, pero sin usar OpenCL. Fuente: Elaborado por el autor.

En dicha tabla se observan los tiempos de procesamiento del código implementado en lenguaje C/C++ usando el esqueleto **SEQSkeleton**, con tres operadores de imagen ejecutados secuencialmente: “Color to Grayscale Converter”, “Sobel’s Edge Detection” y “Rotating Image”, corriendo en un solo núcleo del CPU. Estos tiempos son tomados como referencia en la determinación de la aceleración del cómputo con los esqueletos **PipeSkeleton** y **TaskSkeleton**.

Como resultado cualitativo de la implementación y uso de **SEQSkeleton**, comparando el código 4.9 y 4.11 se puede observar la poca complejidad y brevedad del código de **SEQSkeleton**. El esqueleto logra ocultar y en consecuencia facilitar la implementación en C/C++ de la aplicación de procesamiento de imágenes de prueba. El código fuente de la cabecera (header ”skeletoncore.hpp“) que muestra el detalle de la implementación de la **PAPI SkeletonCoRe** se puede encontrar en el Anexo C, código fuente B.19.

También, el esqueleto **SEQSkeleton** como se observa presenta la misma interfaz de llamada y pase de parámetros que los demás esqueletos paralelos reconfigurables de **SkeletonCoRe**, gestionando internamente la creación del hilo para el procesamiento secuencial y el orden secuencial de las tareas dentro de éste hilo, usando sólo instrucciones nativas del lenguaje de programación C/C++.

Capítulo 5

PipeSkeleton: Un Esqueleto Algorítmico Reconfigurable basado en el Modelo de Cómputo Paralelo *"Pipeline"*

"El logro más impresionante de la industria del software es su continua anulación de los constantes y asombrosos logros de la industria del hardware."

Henry Petroski

5.1 INTRODUCCIÓN

Como prueba de concepto se presenta el diseño, implementación y evaluación de un esqueleto algorítmico reconfigurable, denominado *PipeSkeleton* "Reconfigurable Pipeline Skeleton[2]". Este esqueleto algorítmico es una instancia del catálogo de esqueletos reconfigurables de la interfaz de programación de aplicaciones paralelas (parallel applications programming interface, PAPI) **SkeletonCoRe**.

5.2 ESPECIFICACIÓN Y DESCRIPCIÓN DEL ESQUELETO PIPESKELETON

Este esqueleto implementa el conocido patrón de procesamiento encauzado por etapas denominado *"pipeline"* en el cual un "stream" de elementos de dato fluye a través de una secuencia de etapas (ordenadas como una tuberia) donde en cada una se aplica una operación o tarea a cada elemento del flujo. Este comportamiento puede ser descrito así, $f\colon \alpha \rightarrow \beta$, donde α representa un "stream" de valores de entrada $a_0, a_1, \ldots, a_i, \ldots, a_{n-1}$ que son operados mediante una composición o cadena secuencial de n funciones $f_1 \ldots f_n$ tal que $f_1\colon a_0 \rightarrow \gamma_1, \ldots, f_i\colon \gamma_{i-1} \rightarrow \gamma_i, \ldots, f_n\colon \gamma_{n-1} \rightarrow \beta$. La idea es explotar el paralelismo a partir del procesamiento sobre diferentes elementos del "stream de entrada", siempre que no haya dependencias de datos.

[2]Proceedings of the Advanced Computing Trends Workshops, 9th Latin America High-Performance Computing Conference, CARLA 2022, Porto Alegre, Brazil, September 26–30, 2022. DOI: https://doi.org/10.5281/zenodo.7469164

PipeSkeleton se diseña como una plantilla de alto nivel que se instancia con varios parámetros, entre datos y comportamientos. Para ello, se ha definido una estructura de datos que hemos denominado "Partición de Computación", de hardware y software, como un contenedor del arreglo de tareas y kernels (una por cada etapa del pipeline), el "stream de datos a procesar y sus resultados" y demas información de configuración asociada al ambiente de computación heterogénea. Un ejemplo del uso y parámetros del esqueleto se muestra a continuación:

PipeSkeleton*(CPUpipe, GPUpipe, FPGApipe, tasksList[], in_file, out_file)*,

donde: a) *CPUpipe y GPUpipe:* son las particiones del pipeline en software en CPU y GPU, respectivamente, b) *FPGApipe:* es la partición de hardware en FPGA del pipeline, c) *tasksList:* es el arreglo de apuntadores a cada tarea de cada etapa y que conforman la cadena de tareas del pipeline, y d) *in_file y out_file:* son los archivos desde donde se lee y escribe, respectivamente, el "stream de datos" a procesar y los resultados.

Según sea el caso, el esqueleto *PipeSkeleton* puede gestionar internamente el pipeline a través de dos particiones, una de software repartida entre el CPU y el GPU, y la otra de hardware en FPGA, ambas con tamaños definidos y cuya suma conforman la longitud total del pipeline. A cada etapa del pipeline le corresponde una tarea específica para procesamiento, incluyendo las tareas de entrada y salida. Para el programador es transparente la interfaz de comunicación que interconecta la partición de software en el computador host con la partición de hardware en la tarjeta FPGA.

La asignación de las tareas siempre se realiza desde la primera etapa de la partición de software y hasta la última etapa de la partición de hardware (según la configuración de computación heterogénea). Internamente el esqueleto gestiona la creación del pipeline y un conjunto de asignaciones de la siguiente forma:

a) $softpipe[i] \leftarrow tasksList[i], i = 1 \ldots size\_softpipe$

b) $hardpipe[j] \leftarrow tasksList[size\_softpipe + j], j = 1 \ldots size\_hardpipe$

Esto permite el codiseño integrado de componentes de hardware destinados al FPGA (como co-procesadores) y de componentes de software destinados al microprocesador (como hilos o procesos). Así, es posible explorar espacios de diseño con el fin de intercambiar funcionalidades entre CPU-GPU y FPGA y elegir la combinación con mejor rendimiento.

5.3 DESCRIPCIÓN DEL EXPERIMENTO DE EVALUACIÓN DE PIPESKELETON

En la presente sección se procede a describir la prueba y experimento realizado a uno de los Esqueletos Reconfigurables de la **PAPI SkeletonCoRe** que se han implementado. Aunque en la subsección 4.6.1 del anterior Capítulo 4 se hizo una descripción general de la aplicación de prueba y experimentos para **SkeletonCoRe**, aquí se dan detalles particulares para la evaluación del **Esqueleto Reconfigurable PipeSkeleton**.

El experimento de evaluación consiste en correr el programa paralelo de procesamiento de imágenes implementado con el esqueleto algorítmico reconfigurable **PipeSkeleton** el cual implementa de forma implícita el modelo de cómputo paralelo encauzado o en pipline. Se busca evaluar y comparar el nivel de abstracción, funcionalidad y rendimiento que se obtiene con respecto al programa implementado en OpenCL C/C++ puro que explota paralelismo de forma expresa o explícita.

Con este experimento se busca medir y comparar las métricas de rendimiento del esqueleto paralelo con la impelmentación secuencial del programa. Es decir, se comparan los tiempos de cómputo medidos

de la implementación secuencial en C/C++ puro (sin OpenCL, sobre un solo núcleo del CPU en el Host) con respecto a los valores de tiempo medidos del esqueleto algorítmico corriendo en GPU y FPGA para determinar su aceleración y rendimiento.

Se ha aprovechado el hecho de que los programas generalmente poseen secciones de código o tareas de cómputo que son intensivas en operaciones de entrada/salida y comunicación de datos (I/O bound), y otras tareas intensivas en operaciones de cómputo o cálculo (CPU bound), a la que denominamos Kernels.

Como ya se ha explicado, el programa de procesamiento de imágenes usado para las pruebas ejecuta varios operadores o kernels en pipeline o encauzadamente a una lista de imagenes. Por supuesto, en el modelo de computo pipeline se establece un orden de precedencia de estos operadores de imagen, el cual es fijo, y son los siguientes: → Color to Gray Conversion → Sobel's Edge Detection → Rotating Image →.

Asimismo, la carga de trabajo de prueba está compuesta por cuatro(4) listas de 256 imagenes de diferentes dimensiones: 256x256, 512x512, 1024x1024 y 2048x2048 pixeles, respectivamente, para explotar paralelismo de datos y procesamiento intensivo en el caso de los esqueletos paralelos.

En la Fig. 4.11 a) del Capítulo 4 (subsección 4.6.6) y en la Fig. 5.1 del presente Capítulo 5, se pueden ver el modelo de procesamiento paralelo en pipeline que se usa para ejecutar la aplicación de procesamiento de imágenes para la prueba de concepto, y en la Fig. 5.2 se describen las configuraciones heterogéneas usadas para realizar la prueba de rendimiento y funcionalidad de *PipeSkeleton*.

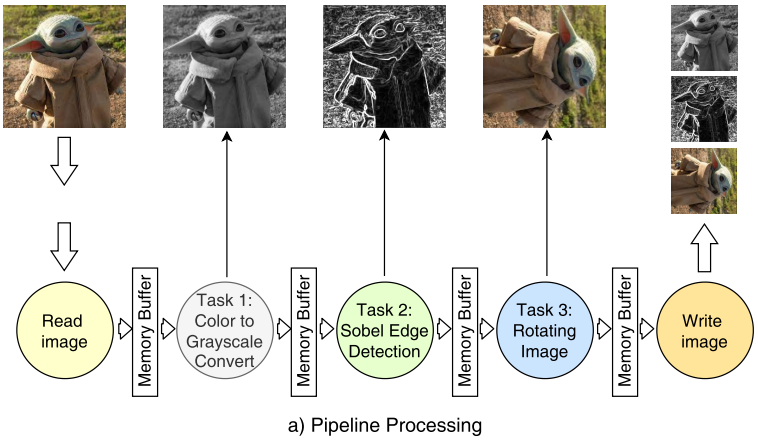


Figura 5.1: Procesamiento de imágenes con cómputo encauzado o pipeline. Fuente: Elaborado por el Autor.

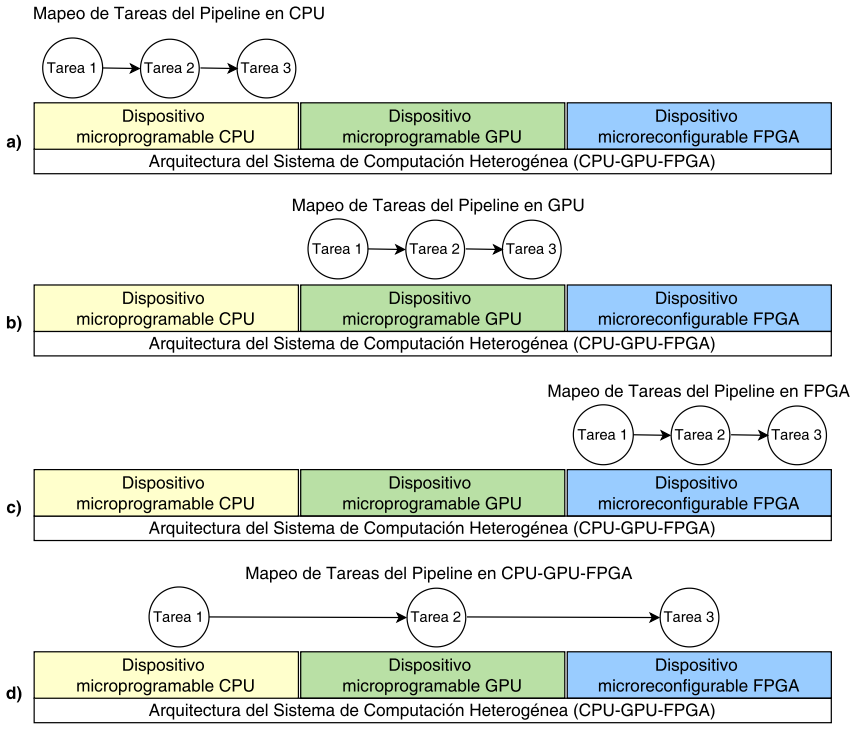


Figura 5.2: Configuración CPU-GPU-FPGA para la Ejecución de la Aplicación de Procesamiento de Imágenes Digitales a Color con PipeSkeleton usando los Dispositivos OpenCL de la Plataforma de Computación Heterogénea. Fuente: Elaborado por el Autor.

Las configuraciones de prueba realizadas se dividen en particiones del pipeline de la forma CPU-GPU-FPGA. Estas configuraciones para el cómputo heterogéneo se han dispuesto así:

- **Configuración A (CPU-0-0):** Se han mapeado o proyectado todo el pipeline como tareas sólo de software en CPU (3-0-0, cpu-0-0), ver la Figura 5.2a,
- **Configuración B (0-GPU-0):** Luego, se han mapeado sólo en GPU (0-3-0, 0-gpu-0), ver la Figura 5.2b,
- **Configuración C (0-0-FPGA):** Después, se ha mapeado todo el pipeline como tareas sólo de hardware en el dispositivo FPGA (0-0-3, 0-0-fpga), ver la Figura 5.2c.
- **Configuración D (CPU-GPU-FPGA):** Por último, se ha mapeado el pipeline en CPU-GPU-FPGA (1-1-1, cpu-gpu-fpga) dividiéndolo en una etapa o kernel por cada dispositivo de cómputo o etapa, como se puede ver en la Figura 5.2d,

Por último, es importante mencionar que en todas las configuraciones de prueba, las tareas de lectura y escritura de datos se han mapeado de manera fija sólo en el Host (CPU).

5.4 IMPLEMENTACIÓN DEL ESQUELETO RECONFIGURABLE PIPESKELETON

A continuación se realiza la codificación en OpenCL C/C++ puro (sin esqueletos), se compila y corre la aplicación paralela que se ha elegido como prueba.

En el código fuente 5.12 se observa la implementación de la aplicación de procesamiento de imágenes sin esqueletos, es decir, programando de forma explícita el paralelismo con instrucciones OpenCL C/C++, note

lo extenso del código fuente del programa expresado en OpenCL C/C++. Luego, en el código fuente 5.13, se muestra la plantilla **PipeSkeleton** del catálogo de esqueletos reconfigurables de **SkeletonCoRe** usada para codificar la misma aplicación paralela para procesamiento de imágenes digitales, note lo breve del código fuente de la misma aplicación de prueba.

```
/*#########################################################################################################
# Autor: Carlos Acosta (25/02/2022)
# Nombre del archivo: opencl-ejemplo-tesis3.cpp
# Función: Ejemplo en openCL/C++ que muestra la estructura de un programa OpenCL.
#          Operador de imagen Sobel Edge Detection para imégenes en formato "jpg".
#########################################################################################################*/
// DECLARACION DE ENCABEZADOS DEL LENGUAJE C/C++
#include <stdio.h>   // Available in both C and C++
#include <iostream>
#include <chrono>
#include <typeinfo>
#include <stdlib.h>
#include <string.h> // Supported both in C and C++
#include <iostream> // Exclusive to C++
#include <vector>   // An exclusive feature of C++
#include <ctime>
#include <time.h>
// DECLARACION DEL ENCABEZADO DE OPENCL C/C++
#ifdef __APPLE__
#include <OpenCL/cl2.h>
#else
#define CL_HPP_TARGET_OPENCL_VERSION 210
#include <CL/cl2.hpp>
#endif

// DEFINICIÓN DE KERNELS
const char *suma_vec =
"__kernel void addVectors(__global const float *a,\n"
" __global const float *b,\n"
" __global float *c) { \n"
"   int gid = get_global_id(0);\n"
"   c[gid] = a[gid] + b[gid];\n"
" }\n";

//############ PROGRAMA PRINCIPAL ##############
int main(int argc, char ** argv) {
  // Dispositivos de cómputo
  const int CPU  = 0;
  const int GPU  = 1;
  const int FPGA = 2;

  std::cout << "-->  0. Iniciando la ejecución del programa!\n";
  //*****************************************************************************************
  //*** 1. Inicializar la plataforma de computación heterogénea.
    // Declaración de variables locales
    std::cout << "-->  1. Inicializando la Plataforma de Computación OpenCL...!\n";
    std::cout << "-->     1.1 Declarando variables para el registro de dispositivos OpenCL de la \
  plataforma heterogénea!\n";
    //
    std::cout << "-->     1.2 Declarando variables para la Carga del Kernel...!\n";
    // Parámetros para carga del Kernel desde el archivo "sobelEdgeDetection.cl"
    FILE  *imputImageFile;
```

Código fuente 5.12: **Programación Paralela Explícita:** Código fuente del Programa de Procesamiento de Imágenes Digitales (ejecutando sólo el algoritmo Sobel Edge Detection).Aquí se muestran explícitamente las instrucciones paralelas del API OpenCL C/C++ que explotan paralelismo sin usar esqueletos paralelos reconfigurables de **SkeletonCoRe**. Fuente: Elaborado por el Autor.

```
53      FILE  *outputImageFile;
54    FILE  *kernelFile;
55    char  *kernelSource;
56    size_t kernelSize;
57      //
58      std::cout << "-->     1.3 Preparando los datos y carga de trabajo de entrada...!\n";
59      // Declaracion, dimensión y reserva de memoria de los vectores de entrada y salida (A, B y C)
60      int SIZE = 1024; // Dimensión de los vectores
61    float *imagenA = (float*)malloc(sizeof(float)*SIZE);  // Vector A de entrada
62    float *imagenB = (float*)malloc(sizeof(float)*SIZE);  // Vector B de entrada
63    float *imagenC = (float*)malloc(sizeof(float)*SIZE);  // Vector C de salida
64
65      // Leyendo las imagenes de entrada A y B.
66    imputImageFile = fopen("inputImage.jpg", "rb");
67
68    //*********************************************************************************************
69    // *** 2. Descubrir la plataforma y los dispositivos (Discovering the platform and devices).
70      std::cout << "-->  2. Descubriendo la Plataforma de Computación Heterogénea...!\n";
71    // Getting platform and device information
72    cl_platform_id platformId    = NULL;
73    cl_device_id   deviceID      = NULL;
74    cl_uint        retNumDevices;
75    cl_uint        retNumPlatforms;
76    cl_int         ret = 0;
77
78    // Se obtiene la cantidad de plataformas en el sistema
79    ret = clGetPlatformIDs(1, &platformId, &retNumPlatforms);
80    std::cout << "Plataforma ID " << ret << "\n";
81
82      // Se obtiene la cantidad de dispositivos en el sistema
83    ret = clGetDeviceIDs(platformId, CL_DEVICE_TYPE_DEFAULT, 1, &deviceID, &retNumDevices);
84    std::cout << "Device     ID " << ret << "\n";
85
86      // Se obtiene la información descriptiva de los dispositivos en el sistema
87    descubriendoDispositivosOpenCL();
88
89    //*** 3. Creación de un contexto para el CPU (Creating a context for CPU).
90      std::cout<<"-->  3. Creando el contexto de procesamiento para el CPU!\n";
91    cl_context cpu_context = clCreateContext(NULL, 1, &deviceID, NULL, NULL,  &ret);
92
93    //*** 4. Creación de una cola de comandos para el dispositivo CPU (Creating a command-queue for CPU).
94    std::cout<<"-->  4. Creando cola de comandos por cada dispositivo de cómputo!\n";
95    cl_command_queue commandQueue = clCreateCommandQueue(cpu_context, deviceID, 0, &ret);
96
97    //*** 5. Creación de objetos de memoria (buffers) para contener datos (Creating memory objects
98    // (buffers) to hold data).
99    std::cout<<"-->  5. Creando objetos de memoria para datos de entrada!\n";
100   cl_mem aMemObj = clCreateBuffer(cpu_context, CL_MEM_READ_ONLY, SIZE * sizeof(float), NULL, &ret);
101   cl_mem bMemObj = clCreateBuffer(cpu_context, CL_MEM_READ_ONLY, SIZE * sizeof(float), NULL, &ret);
102   cl_mem cMemObj = clCreateBuffer(cpu_context, CL_MEM_WRITE_ONLY, SIZE * sizeof(float), NULL, &ret);
103
104   //*** 6. Copia de los datos de entrada en el dispositivo (Copying the input data onto the device,
105   // Copy lists to memory buffers).
106   std::cout<<"-->  6. Copiando datos de trabajo a cada dispositivo de cómputo!\n";
107   ret = clEnqueueWriteBuffer(commandQueue, aMemObj, CL_TRUE, 0, SIZE * sizeof(float), imagenA, 0, NULL, NULL);
108   ret = clEnqueueWriteBuffer(commandQueue, bMemObj, CL_TRUE, 0, SIZE * sizeof(float), imagenB, 0, NULL, NULL);
109
110   //*** 7. Creación y compilación de un programa a partir del código fuente OpenCL C/C++ (Creating and compiling a program
```

(Cont. Codigo fuente 5.12) **Programación Paralela Explícita:** Código fuente del Programa de Procesamiento de Imágenes Digitales (ejecutando sólo el algoritmo Sobel Edge Detection).Aquí se muestran explícitamente las instrucciones paralelas del API OpenCL C/C++ que explotan paralelismo sin usar esqueletos paralelos reconfigurables de **SkeletonCoRe**. Fuente: Elaborado por el Autor.

```
111   std::cout<<"-->  7. Creando y compilando el programa OpenCL!\n";
112   kernelFile = fopen("sobelEdgeDetection.cl", "r");
113   if (!kernelFile) {
114     fprintf(stderr, "No file named sobelEdgeDetection.cl was found\n");
115     exit(-1);
116   }
117   kernelSource = (char*)malloc(MAX_SOURCE_SIZE);
118   kernelSize   = fread(kernelSource, 1, MAX_SOURCE_SIZE, kernelFile);
119   fclose(kernelFile);
120
121     // Create program from kernel source
122   cl_program program = clCreateProgramWithSource(cpu_context, 1, (const char **)&kernelSource, \
123   (const size_t *)&kernelSize, &ret);
124
125     // Build program
126   ret = clBuildProgram(program, 1, &deviceID, NULL, NULL, NULL);
127
128   //*** 8. Extracción de Kernels del programa (Extracting the kernel from the program, Create kernel).
129   std::cout<<"-->  8. Estracción de Kernels de programa OpenCL!\n";
130   cl_kernel kernel = clCreateKernel(program, "addVectors", &ret);
131   // Set arguments for kernel
132   ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&aMemObj);
133   ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&bMemObj);
134   ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&cMemObj);
135
136   //*** 9. Ejecución de Kernels (Executing the kernel).
137   std::cout<<"-->  9. Ejecutando Kernels...";
138   // Execute the kernel
139   size_t globalItemSize = SIZE;
140   size_t localItemSize  = 64; // globalItemSize has to be a multiple of localItemSize. 1024/64 = 16
141   ret = clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL, &globalItemSize, &localItemSize,\
142           0, NULL, NULL);
143
144   //*** 10. Copia de los datos de salida al Host (Copying output data back to the Host).
145   std::cout<<"--> 10. Copiando datos de salida hacia el Host o Anfitrión!\n";
146
147     // Read from device back to host.
148   ret = clEnqueueReadBuffer(commandQueue, cMemObj, CL_TRUE, 0, SIZE * sizeof(float), C, 0, NULL, NULL);
149
150   grabarResultados(imagenA, imagenB, imagenC, SIZE);
151
152   //*** 11. Liberación de los recursos OpenCL (Releasing the OpenCL resources).
153   std::cout<<"--> 11. Liberando los recursos OpenCL asignados a los dispositivos de cómputo!\n";
154   // Clean up, release memory.
155   ret = clFlush(commandQueue);
156   ret = clFinish(commandQueue);
157   ret = clReleaseCommandQueue(commandQueue);
158   ret = clReleaseKernel(kernel);
159   ret = clReleaseProgram(program);
160   ret = clReleaseMemObject(aMemObj);
161   ret = clReleaseMemObject(bMemObj);
162   ret = clReleaseMemObject(cMemObj);
163   ret = clReleaseContext(cpu_context);
164   free(imagenA);
165   free(imagenB);
166   free(imagenC);
167
168   std::cout<<"--> 12. Culminó con éxito el programa!\n";
169   return 0; //FIN DEL PROGRAMA
170 }
```

(Cont. Codigo fuente 5.12) **Programación Paralela Explícita:** Código fuente del Programa de Procesamiento de Imágenes Digitales (ejecutando sólo el algoritmo Sobel Edge Detection).Aquí se muestran explícitamente las instrucciones paralelas del API OpenCL C/C++ que explotan paralelismo sin usar esqueletos paralelos reconfigurables de **SkeletonCoRe**. Fuente: Elaborado por el Autor.

```
171 //######################### DEFINICIÓN DE FUNCIONES ########################
172 /*** Función que detecta las plataformas y sus dispositivos de cómputo OpenCL disponibles ***/
173 void descubriendoDispositivosOpenCL(void) {
174     // DECLARACION DE VARIABLES LOCALES
175     char queryBuffer[1024];
176     cl_int clError;
177 
178     // Get the Number of Platforms available
179     // Note that the second parameter "platforms" is set to NULL. If this is NULL then this
180   // argument is ignored
181     // and the API returns the total number of OpenCL platforms available.
182   cl_platform_id *platforms     = NULL;
183     cl_uint         num_platforms = 0;
184 
185   // Se reserva memoria para el arreglo de plataformas
186   platforms = (cl_platform_id *)malloc(sizeof(cl_platform_id)*num_platforms);
187     if ((clGetPlatformIDs(0, NULL, &num_platforms)) == CL_SUCCESS) {
188         printf("        2.1 Número de Plataformas disponibles en el sistema heterogéneo: %d\n", \
189     num_platforms);
190         platforms = (cl_platform_id *)malloc(sizeof(cl_platform_id)*num_platforms);
191         if(clGetPlatformIDs(num_platforms, platforms, NULL) != CL_SUCCESS) {
192             free(platforms);
193             printf("Error in call to clGetPlatformIDs....\n Exiting");
194             exit(0);
195         }
196     }
197     if (num_platforms == 0) {
198         printf("No OpenCL Platforms Found ....\n Exiting");
199         exit(0);
200     }
201     else {
202         // We have obtained one platform here.
203         // Lets enumerate the devices available in this Platform.
204         for (cl_uint pIndex = 0; pIndex < num_platforms; pIndex++) {
205       cl_device_id    *devices;
206             cl_uint          num_devices;
207 
208             clError = clGetDeviceIDs (platforms[pIndex], CL_DEVICE_TYPE_ALL, 0, NULL, &num_devices);
209       printf("        2.2 Plataforma ID(%d): %d ", pIndex, num_devices);
210             if (clError != CL_SUCCESS) {
211                 printf("Error Getting number of devices... Exiting\n ");
212                 exit(0);
213             }
214             // If successfull the num_devices contains the number of devices available in the platform
215             // Now lets get all the device list. Before that we need to malloc devices
216       devices = (cl_device_id *)malloc(sizeof(cl_device_id) * num_devices);
217             clError = clGetDeviceIDs (platforms[pIndex], CL_DEVICE_TYPE_ALL, num_devices, devices,\
218                  &num_devices);
219             if (clError != CL_SUCCESS) {
220                 printf("Error Getting number of devices... Exiting\n ");
221                 exit(0);
222             }
223 
224             for (cl_uint dIndex = 0; dIndex < num_devices; dIndex++) {
225                 //imprimeInfoDispositivo(devices[dIndex]);
226         // Se obtiene el Nombre del Dispositivo de Cómputo OpenCL
227           clError = clGetDeviceInfo(devices[dIndex], CL_DEVICE_NAME, sizeof(queryBuffer),\
228                     &queryBuffer, NULL);
229         printf("Dispositivo(s) OpenCL (ID %d): %s ", dIndex, queryBuffer); //CL_DEVICE_NAME
230           queryBuffer[0] = '\0';
```

(Cont. Codigo fuente 5.12) **Programación Paralela Explícita:** Código fuente del Programa de Procesamiento de Imágenes Digitales (ejecutando sólo el algoritmo Sobel Edge Detection).Aquí se muestran explícitamente las instrucciones paralelas del API OpenCL C/C++ que explotan paralelismo sin usar esqueletos paralelos reconfigurables de **SkeletonCoRe**. Fuente: Elaborado por el Autor.

```
231         clError = clGetDeviceInfo(devices[dIndex], CL_DEVICE_VERSION, sizeof(queryBuffer),\
232                     &queryBuffer, NULL);
233         printf("--> Versión %s\n", queryBuffer); //CL_DEVICE_VERSION
234           queryBuffer[0] = '\0';
235             }
236         }
237     }
238 }
239
240 /*** Función que graba en un archivo los resultados del procesamiento ***/
241 void grabarResultados(float *vec_A, float *vec_B, float *vec_C, int tamVec) {
242   // Declaración de variables locales
243     FILE *archivoSal;
244   char nombreArchivoSal[] = "archivo_salida.dat";
245
246     archivoSal = fopen("archivo_salida.dat", "w+"); // Nombre FILE y modo escritura: "w"
247     if(archivoSal==NULL) {
248         printf("Error creando el archivo de salida %s:\n", nombreArchivoSal);
249         exit(1);
250     }
251   // Grabando vectores en archivo de salida
252   for (int i = 0; i < tamVec; ++i) {
253     fprintf(archivoSal, "%f + %f = %f\n", vec_A[i], vec_B[i], vec_C[i]);
254   }
255     fclose(archivoSal); // Cerrando el archivo de salida
256   std::cout << "        10.2 ==> Resultados grabados con éxito en archivo de salida: "
257             << nombreArchivoSal;
258 }
259
260 /*** Función que imprime la información asociada a cada dispositivo de cómputo OpenCL ***/
261 void imprimeInfoDispositivo(cl_device_id device) {
262   // Declaración de variables locales
263     char queryBuffer[1024];
264     cl_int clError;
265
266     /* Se obtiene el Nombre del Dispositivo de Cómputo OpenCL */
267     clError = clGetDeviceInfo(device, CL_DEVICE_NAME, sizeof(queryBuffer), &queryBuffer, NULL);
268   printf("Dispositivo(s) OpenCL: %s ", queryBuffer); //CL_DEVICE_NAME
269     queryBuffer[0] = '\0';
270
271     /* Se obtiene la Version OpenCL del Dispositivo de Cómputo */
272     clError = clGetDeviceInfo(device, CL_DEVICE_VERSION, sizeof(queryBuffer), &queryBuffer, NULL);
273   printf("--> Versión %s\n", queryBuffer); //CL_DEVICE_VERSION
274     queryBuffer[0] = '\0';
275
276 }
277 //#############################################################################################
278 //# SOBEL EDGE DETECTION KERNEL en OpenCL/C++ que contiene el Código de Cómputo Intensivo
279 //#############################################################################################
280 // Runs the Sobel Filter on an image, this expects the image to be read in using unsigned integers to
281 // represent the RGB channels and should be between 0 and 255. This means the format CL_UNSIGNED_INT8
282 // should be used when creating the image2d_t source and output
283 __kernel void sobelEdgeDetection(
284   __read_only image2d_t sourceImage,
285   __write_only image2d_t outputImage,
286   int width,
287   int height
288   )
289 {
290   // This is the currently focused pixel and is the output pixel
```

(Cont. Codigo fuente 5.12) **Programación Paralela Explícita:** Código fuente del Programa de Procesamiento de Imágenes Digitales (ejecutando sólo el algoritmo Sobel Edge Detection).Aquí se muestran explícitamente las instrucciones paralelas del API OpenCL C/C++ que explotan paralelismo sin usar esqueletos paralelos reconfigurables de **SkeletonCoRe**. Fuente: Elaborado por el Autor.

```
291   // location
292   int2 ImageCoordinate = (int2)(get_global_id(0), get_global_id(1));
293
294   // Make sure we are within the image bounds
295   if (ImageCoordinate.x < width && ImageCoordinate.y < height) {
296     int x = ImageCoordinate.x;
297     int y = ImageCoordinate.y;
298
299     // Read the 8 pixels around the currently focused pixel
300     uint4 Pixel00 = read_imageui(sourceImage, Sampler, (int2)(x - 1, y - 1));
301     uint4 Pixel01 = read_imageui(sourceImage, Sampler, (int2)(x, y - 1));
302     uint4 Pixel02 = read_imageui(sourceImage, Sampler, (int2)(x + 1, y - 1));
303
304     uint4 Pixel10 = read_imageui(sourceImage, Sampler, (int2)(x - 1, y));
305     uint4 Pixel12 = read_imageui(sourceImage, Sampler, (int2)(x + 1, y));
306
307     uint4 Pixel20 = read_imageui(sourceImage, Sampler, (int2)(x - 1, y + 1));
308     uint4 Pixel21 = read_imageui(sourceImage, Sampler, (int2)(x, y + 1));
309     uint4 Pixel22 = read_imageui(sourceImage, Sampler, (int2)(x + 1, y + 1));
310
311     // This is equivalent to looping through the 9 pixels under this convolution and applying
312     // the appropriate filter, here we've already applied the filter coefficients since they are static
313     uint4 Gx = Pixel00 + (2 * Pixel10) + Pixel20 -
314         Pixel02 - (2 * Pixel12) - Pixel22;
315
316     uint4 Gy = Pixel00 + (2 * Pixel01) + Pixel02 -
317         Pixel20 - (2 * Pixel21) - Pixel22;
318
319     // Holds the output RGB values
320     uint4 OutColor = (uint4)(0, 0, 0, 1);
321
322     // Compute the gradient magnitude
323     OutColor.x = sqrt((float)(Gx.x * Gx.x + Gy.x * Gy.x)); // R
324     OutColor.y = sqrt((float)(Gx.y * Gx.y + Gy.y * Gy.y)); // G
325     OutColor.z = sqrt((float)(Gx.z * Gx.z + Gy.z * Gy.z)); // B
326
327     // Adjust all of the RGB values to not be more than 255
328     if (OutColor.x > 255) {
329       OutColor.x = 255;
330     }
331     if (OutColor.y > 255) {
332       OutColor.y = 255;
333     }
334     if (OutColor.z > 255) {
335       OutColor.z = 255;
336     }
337     // Convert to grayscale using luminosity method
338     uint Gray = (OutColor.x * 0.2126) + (OutColor.y * 0.7152) + (OutColor.z * 0.0722);
339
340     // Write the RGB value to the output image
341     write_imageui(outputImage, ImageCoordinate, (uint4)(Gray, Gray, Gray, 0));
342   }
343 }
```

(Cont. Codigo fuente 5.12) **Programación Paralela Explícita:** Código fuente del Programa de Procesamiento de Imágenes Digitales (ejecutando sólo el algoritmo Sobel Edge Detection).Aquí se muestran explícitamente las instrucciones paralelas del API OpenCL C/C++ que explotan paralelismo sin usar esqueletos paralelos reconfigurables de **SkeletonCoRe**. Fuente: Elaborado por el Autor.

En el código fuente 5.13 se muestra la plantilla de **PipeSkeleton** usando los objetos y construcciones del **PAPI SkeletonCoRe** para encapsular el paralelismo y permitir la programación paralela implícita, con base en el API OpenCL en lenguaje C/C++. Además, en el código fuente en 5.14 se presenta un

ejemplo de programa principal donde se usa el esqueleto. Con este programa principal se compila, corre y prueba la aplicación paralela para procesamiento de imágenes digitales con **PipeSkeleton**. En éste se explota paralelismo implícito sobre una plataforma de cómputo heterogéneo basada en CPU, GPU y FPGA.

```
//*************************************************************************************************
//* Program name: PipeSkeleton of PAPI SkeletonCORE
//* Programmer  : Carlos Acosta-León
//* Function    : PipeSkeleton Body.
//* Language/API: openCL C/C++
//*************************************************************************************************
//*** Definition of Tasks Function +++//
void Task1_convertImageGrayscale(unsigned char *inImage, unsigned char *outImage[], int width, int height,
              int *channels);
void Task2_sobelEdgeDetection(unsigned char *inImage, unsigned char *(outImage[]), int iWidth, int iHeight,
              int channels);
void Task3_imageRotated(unsigned char *inImage, unsigned char *(outImage[]), int width, int height,
              int channels);
//*** Define an Array with pointer to functions: Pointer Array to Processing Functions
void (*tasksArray[MAXSIZE_TASKSLIST])(unsigned char *inImage, unsigned char *outImage[], int width,
    int height, int *channels, double &exec_time) = {
      *Task1_convertImageGrayscale(),
      *Task2_sobelEdgeDetection(),
      *Task3_imageRotated(),
  };
/*** PipeSkeleton Skeleton Body for Pipeline Parallel Processing ***/
void pipeSkeleton(skeletoncore::devicePartition partition1, skeletoncore::devicePartition partition1,
          skeletoncore::devicePartition partition1, void (*tasksArray[])(void), int num_tasks) {
    //* Data
  workload_size = 256;
  //* Create Communication Channels between devices CPU, GPU and FPGA
  channel float cpu_gpu_channel, gpu_fpga_channel, fpga_cpu_channel;
  //* Get the device(s)
  err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &devices[1], &partition1.num_devices);
  err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &devices[2], &partition2.num_devices);
  err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_FPGA, 1, &devices[3], &partition3.num_devices);
  //* Create contexts for devices CPU, GPU and FPGA
  cl_context context;
  context = clCreateContext(0, 3, devices, NULL, NULL, NULL, &err);
  //* Create commands queue for devices
  cl_command_queue queue_gpu, queue_cpu, queue_fpga;
  partition1.device_cpu  = clCreateCommandQueue(context, devices[1], 0, &err);
  partition2.device_gpu  = clCreateCommandQueue(context, devices[2], 0, &err);
  partition3.device_fpga = clCreateCommandQueue(context, devices[3], 0, &err);
  //* Basic pipeline for PieSkeleton
  //*** Executing on CPU
  __kernel void device_cpu(__global int* in) {
    for (int i = 0; i < workload_size; ++i) {
        int *image_in = read_ListImage();         // Get images by segments
        image_out[partition1.dataSet.images.properties.width * partition1.dataSet.images.properties.height]
          = partition1.kernel.(*tasksArray[1])(&image_in, 0, width*height, channelRGB); // do some work
        write_channel(cpu_gpu_channel, &image_out);    // send data to the next partition
      }
  }
  //*** Executing on GPU
  __kernel void device_gpu() {
    for (int i = 0; i < workload_size; ++i) {
      int *image_in = read_channel(cpu_gpu_channel);  // take data from cpu
      image_out[partition2.dataSet.images.properties.width * partition2.dataSet.images.properties.height]
          = partition2.kernel.(*tasksArray[2])(&image_in, 0, width*height, channelRGB); // do some work
```

Código fuente 5.13: **Programación Paralela Implícita:** Código que muestra la implementación en openCL C/C++ del Esqueleto Reconfigurable **PipeSkeleton**. El detalle del código se puede encontrar en el Anexo C, código fuente B.19. Fuente: Elaborado por el autor.

```
56        write_channel(gpu_fpga_channel, &image_out);    // send data to the next partition
57      }
58    }
59    //*** Executing on FPGA
60    __kernel void device_fpga(__global int* out) {
61      for (int i = 0; i < workload_size; ++i) {
62        int *image_in = read_channel(gpu_fpga_channel);         // take data from gpu
63        image_out[partition3.dataSet.images.properties.width * partition3.dataSet.images.properties.height]
64            = partition3.kernel.(*tasksArray[2])(&image_in, 0, width*height, channelRGB); // do some work
65        write_channel(fpga_cpu_channel, &image_out);            // write result in the end
66      }
67    }
68    //* Start concurrently tasks (kernels) for partitions in CPU-GPU-FPGA
69    clEnqueueTask(device_cpu, *partition1.kernel.(*tasksArray[1]));
70    clEnqueueTask(device_gpu, *partition2.kernel.(*tasksArray[2]));
71    clEnqueueTask(device_fpga,*partition3.kernel.(*tasksArray[3]));
72    clFinish(device_fpga); // last kernels in our pipeline
73  }
```

(Cont. Codigo fuente 5.13) **Programación Paralela Implícita:** Código que muestra la implementación en OpenCL C/C++ del Esqueleto Reconfigurable PipeSkeleton. El detalle del código se puede encontrar en el Anexo C, código fuente B.19. Fuente: Elaborado por el autor.

En el código fuente mostrado en 5.14 se puede ver un ejemplo donde se muestra un Programa Principal que usa el Esqueleto Reconfigurable PipeSkeleton y demás componentes de la **PAPI SkeletonCoRe**.

```
1  //************************************************************************************************
2  //* Program name: Library of Objects and Parallel Skeletons of PAPI SkeletonCORE
3  //* Programmer  : Carlos A. Acosta-León
4  //* Function    : Show the PipeSkeletons Usage.
5  //* Language/API: OpenCL C/C++
6  //* Date        : 25/02/2022
7  //************************************************************************************************
8  #include "papi-skeletoncore-header_ver_1-4.hpp"
9  //* Define User's Tasks or Functions for processing
10 #include "user_define_tasks_body_ver_1-4.hpp"
11 #define BILLION 1000000000L;
12 const   int MAXSIZE_TASKSLIST = 7;
13
14 // Declare space of names for SkeletonCoRe
15 using namespace skeletoncore_papi;
16 //using namespace user_tasks_body;
17 using namespace std;
18
19 /******* BEGIN MAIN PROGRAM ***************************************************/
20 int main(int argc, char *argv[]) {
21     //* Declare data variables
22     string  imageType     = "jpg"; // type of jpg, png, etc, for example
23     string  mytasksPool[3];
24     int     numTasks = 3;
25     //* FIRST STEP: DECLARE INSTANCES OF SKELETONCORE OBJECTS
26     //* Declare skeletoncore objects
27     skeletoncore::papi_skeletoncore skelcore;  // Create the setup enviorement
28     skeletoncore::dataSet           mydataSet; // Create container for images data/operations
29     // device Partition for CPU, GPU and FPGA Processing
30     skeletoncore::devicePartition   myCPUPartition, myGPUPartition, myFPGAPartition;
31     //* SECOND STEP: SETUP THE SKELETONCORE ENVIORENMENT AND COMPUTING PARTITIONS
```

Código fuente 5.14: **Programación Paralela Implícita:** Código del Programa Principal que usa el Esqueleto **PipeSkeleton** de la **PAPI SkeletonCoRe** para explotar paralelismo de forma implícita con CPU, GPU y FPGA. El código fuente de la cabecera (header "papi-skeletoncore.hpp") se puede encontrar en el Anexo C, código fuente B.19. Fuente: Elaborado por el autor.

```
32      //* Initialize the skeletoncore processing enviorement
33      skelcore.init(argc);
34      //* Getting info of computing devices in platform
35      skelcore.platformDiscovery(skelcore.deviceGroup);
36      //* Getting input data: Here it gests all properties from input image readed
37      mydataSet.images.readDataSetImage("jpg", argv[1], mydataSet, &mydataSet.images.input_Images);
38      //* Definition of Tasks body: Must be inside the "user's_define_tasks_body.hpp" header file
39      skelcore.getTasksKernel(mytasksPool);
40      //* THIRD STEP: SETUP THE COMPUTING PARTITIONS
41      //* Setting Up Partitions for compute devices
42      skelcore.partitionSetup(skelcore.deviceGroup, myCPUPartition, mydataSet.images.cpu,  mytasksPool[0]);
43      skelcore.partitionSetup(skelcore.deviceGroup, myGPUPartition, mydataSet.images.gpu,  mytasksPool[1]);
44      skelcore.partitionSetup(skelcore.deviceGroup, myFPGAPartition, mydataSet.images.fpga,mytasksPool[2]);
45    //* FOURTH STEP: EXECUTE PIPESKELETON PARALLEL SKELETON FROM SKELETONCORE PAPI
46      skelcore.exec.pipeSkeleton(myCPUPartition, myGPUPartition, myFPGAPartition, mytasksPool[0], numTasks);
47      //* FIFTH STEP: GET AND WRITE RESULTS AFTER PROCESSING
48      //* Metrics of execution time
49      skelcore.getExecTime(myCPUPartition, myGPUPartition, myFPGAPartition);
50      //* Putting out results into Files
51      mydataSet.images.writeOutputImages("image-out-list.jpg", mydataSet.images.properties.width,
52        mydataSet.images.properties.height, mydataSet.images.properties.channels, mydataSet.images.out_);
53      //* SIXTH STEP: CLOSING SKELETONCORE ENVIORENMENT
54      skelcore.terminate();
55      return 0; /*** END MAIN PROGRAM ***/
56  }
```

(Cont. Codigo fuente 5.14). **Programación Paralela Implícita:** Código del Programa Principal que usa el Esqueleto **PipeSkeleton** de la **PAPI SkeletonCoRe** para explotar paralelismo de forma implícita con CPU, GPU y FPGA. El código fuente de la cabecera (header "papi-skeletoncore.hpp") se puede encontrar en el Anexo C, código fuente B.19. Fuente: Elaborado por el autor.

Al compilar y correr el programa en OpenCL/C++ de procesamiento de imagen con PipeSkeleton mostrado en el código fuente 5.14 obtenemos la siguiente traza de salida resultado de las diferentes acciones que se realizan durante la preparación, configuración y procesamiento en la plataforma heterogénea:

```
$> g++ pipeskeleton.cpp -o pipeskeleton.out -lOpenCL
$> ./pipeskeleton.out
===>  ******* Beginning of the SKELETONCORE PAPI Environment *******
--> 00.   Discovering the Opencl Platform and Devices...!
00.1  OpenCL Platforms Available on this Heterogeneous System: 2
00.2  Plataform ID=0 --> OpenCL Divice detected: Intel(R) Core(TM) i5-10400F CPU @ 2.90GHz
00.2  Plataform ID=1 --> OpenCL Divice detected: NVIDIA GeForce GTX 1060 3GB
00.2  Plataform ID=2 --> OpenCL Divice detected: Intel Stratix 10 FPGA
--> 01. ==>Leyendo Datos de la Carga de Trabajo a Procesar...!
--> 02. ==>Configurando la Partición de Cómputo del: CPU
02.1  Creando su contexto de procesamiento
02.2  Creando la cola de comandos del CPU
02.3  Creando objetos de memoria para datos
02.4  Copiando los datos al Dispositivo de Cómputo
03.1  Configurando el Kernel asociado a la Partición...
03.2  Creando y compilando el programa y Kernel OpenCL!
03.2  Extracción de Kernels de programa OpenCL
-->  2.   Leyendo Vectores de Datos de la Carga de Trabajo a Procesar...!
--> 02.   ==>Configurando la Partición de Cómputo del: GPU
02.1  Creando su contexto de procesamiento
02.2  Creando la cola de comandos del GPU
02.3  Creando objetos de memoria para datos
02.4  Copiando los datos al Dispositivo de Cómputo
03.1  Configurando el Kernel asociado a la Partición...
03.2  Creando y compilando el programa y Kernel OpenCL!
03.2  Extracción de Kernels de programa OpenCL
--> 02. ==>Configurando la Partición de Cómputo del: FPGA
02.1  Creando su contexto de procesamiento
```

```
02.2  Creando la cola de comandos del FPGA
02.3  Creando objetos de memoria para datos
02.4  Copiando los datos al Dispositivo de Cómputo
03.1  Configurando el Kernel asociado a la Partición...
03.2  Creando y compilando el programa y Kernel OpenCL!
03.2  Extracción de Kernels de programa OpenCL
--> 04.  Executing Parallel Skeleton on CPU
--> 05.   Copiando resultados de salida hacia el Host o Anfitrión!
--> 04.  Executing Parallel Skeleton on GPU
--> 05.   Copiando resultados de salida hacia el Host o Anfitrión!
--> 04.  Executing Parallel Skeleton on FPGA
--> 05.   Copiando resultados de salida hacia el Host o Anfitrión!
--> 06.  ===> TIEMPOS DE COMPUTACIÓN:
06.1  CPU  Time: 71321 ns
06.2  GPU  Time:  1707 ns
06.2  FPGA Time:   270 ns
07.2  ===> RESULTADOS GRABADOS EN: archivo_salida_cpu.dat
07.2  ===> RESULTADOS GRABADOS EN: archivo_salida_gpu.dat
07.2  ===> RESULTADOS GRABADOS EN: archivo_salida_fpga.dat
===>  ******* Ending of the SKELETONCORE PAPI Environment *******
```

5.5 EVALUACIÓN DE FUNCIONALIDAD Y RENDIMIENTO DE PIPESKELETON

Como resultado del experimento de prueba, en la Figura 5.3 se observan como ejemplo las imágenes esperadas, producto del procesamiento digital de una imagen de la lista de imágenes originales a color procesadas. Estas se han obtenido a partir de la conversión de las imágenes a color a imágenes en escala de grises, otras imágenes con detección de bordes de los objetos en la imagen usando el algoritmo de Sobel, y las últimas imágenes rotadas 90 grados hacia la derecha (en sentido del reloj).

a) Imagen de entrada original (1024x1024). b) Imagen original Convertida a Escala de Grises (1024x1024). c) Imagen con Detección de Bordes con Sobel (1024x1024). d) Imagen original rotada 90 grados (1024x1024).

Figura 5.3: Imagenes resultantes del procesamiento digital usando el esqueleto **PipeSkeleton**: a) Imagen original a color RGB (formato .jpg), b) Imagen convertida a escala de 256 grises, c) Imagen con bordes de objetos resaltados (con máscara Sobel), y d) Imagen con Rotación. Fuente: Elaborado por el autor.

5.5.1 *RESULTADOS CUANTITATIVOS DE LA PRUEBA DE RENDIMIENTO*

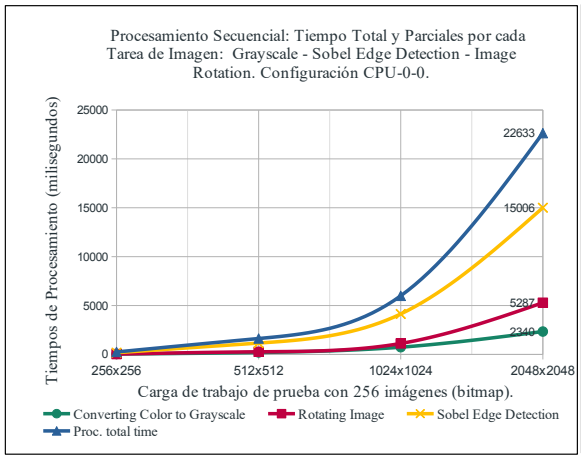
Los resultados cuantitativos de las pruebas se muestran en la Tabla 5.4 (a). Aquí se presentan los tiempos de procesamiento, comunicación y entrada/salida de cuatro listas de 256 imágenes de dimensiones 256x256, 512x512, 1024x1024 y 2048x2048 pixeles, siendo cada imagen una unidad individual de procesamiento. En dicha Tabla se observan los tiempos de ejecución de **PipeSkeleton** corriendo sólo en CPU (configuración A, 3-0-0, las tres tareas corriendo en CPU), luego sólo en GPU (configuración B, 0-3-0, las tres tareas corriendo en GPU), después en FPGA (configuración C, 0-0-3, las tres tareas corriendo en FPGA) y finalmente la combinación de las particiones hardware/software entre CPU, GPU y FPGA (configuración D, 1-1-1, una tarea corriendo en cada dispositivo).

Tabla 5.4: Tiempos de procesamiento usando PipeSkeleton sobre CPU, GPU y FPGA.

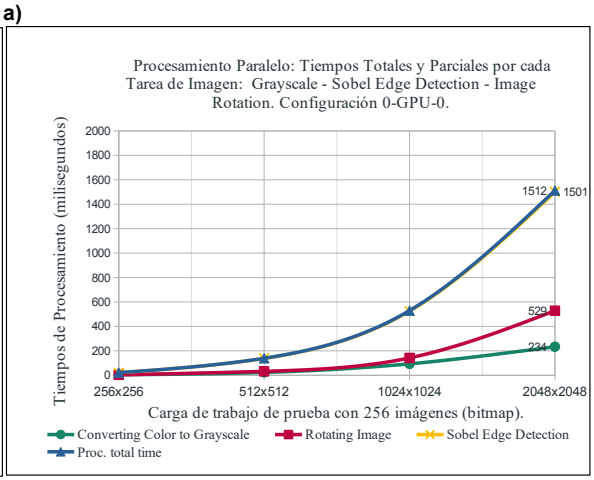
OPENCL, USANDO ESQUELETOS ALGORÍTMICOS: PIPESKELETON

Configuración CPU-GPU-FPGA	Carga de trabajo (Workload)		Tarea 1: Color to Grayscale Convert	Tarea 2: Sobel Edge detection	Tarea 3: Color Image Rotation	Tiempo Total de Proc.	Tiempo de Comunicac. entre Procesos	Tiempo lectura archivos (Entrada)	Tiempo escritura archivos (Salida)	Tiempo Total de Entrada/ Salida	Tiempo Total de Ejecución	Aceleración (SpeedUp)
Partición de Cómputo	Cantidad de Imagenes (Carga de Trabajo)	Dimensión NxN o Tamaño de Imagen (pixels)	Tiempo proc. Color to Grayscale x 256 imágenes (miliseg)	Tiempo proc. Sobel Edge Detection x 256 imágenes (miliseg)	Tiempo proc. Image Rotation x 256 imágenes (miliseg)	Tiempo CPU-GPU-FPGA x 256 imágenes (milisegs.)	Tiempo de Comunic. CPU-GPU-FPGA (milisegs.)	Tiempo de lectura x 256 imágenes (miliseg.)	Tiempo de grabar x 256 imágenes (milisegs.)	Tiempo Total Comunicación x 256 imágenes (miliseg)	Tiempo Proc. + Tiempo Comunic. + Tiempo E/S(miliseg)	Tsec/Tpar
3-0-0, CPU Caso Secuencial en C/C++, sin OpenCL	256	256x256	38,53	179,10	33,05	250,67	0,00	197,40	508,12	705,52	956,19	1,00
		512x512	198,54	1159,01	268,46	1626,01	0,00	835,19	1915,41	2750,59	4376,61	1,00
		1024x1024	738,08	4137,25	1119,17	5994,49	0,00	2631,02	5148,43	7779,46	13773,95	1,00
		2048x2048	2339,54	15006,18	5286,94	22632,67	0,00	7365,16	21726,08	29091,24	51723,91	1,00
0-3-0, GPU	256	256x256	6,09	28,30	5,04	39,42	37,09	197,40	508,12	395,73	472,24	6,36
		512x512	30,27	176,71	42,42	249,40	148,34	835,19	1915,41	1542,81	1940,55	6,52
		1024x1024	110,79	621,00	167,99	899,77	593,36	2631,02	5148,43	4363,50	5856,63	6,66
		2048x2048	378,89	2430,25	856,22	3665,36	2373,45	7365,16	21726,08	16317,28	22356,08	6,17
1-1-1, CPU-GPU-FPGA	256	256x256	7,79	36,22	6,68	50,70	44,40	197,40	508,12	473,76	568,85	4,94
		512x512	36,23	211,51	48,99	296,73	177,59	835,19	1915,41	1847,02	2321,35	5,48
		1024x1024	132,36	741,93	200,70	1074,99	710,36	2631,02	5148,43	5223,91	7009,26	5,58
		2048x2048	493,48	3165,25	1115,17	4773,91	2841,45	7365,16	21726,08	19534,77	27150,13	4,74
0-0-3, FPGA	256	256x256	2,68	12,45	2,30	17,43	18,28	197,40	508,12	195,08	230,78	14,38
		512x512	13,33	77,83	18,03	109,19	73,13	835,19	1915,41	760,54	942,85	14,89
		1024x1024	47,23	264,74	71,62	383,59	292,50	2631,02	5148,43	2151,02	2827,11	15,63
		2048x2048	147,86	948,39	334,13	1430,38	1170,01	7365,16	21726,08	8043,73	10644,12	15,82

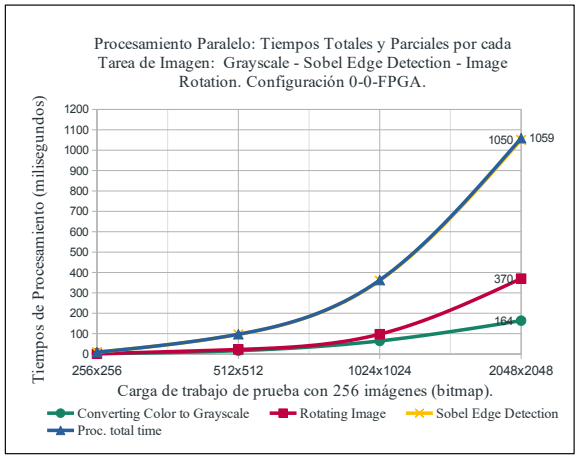
a)



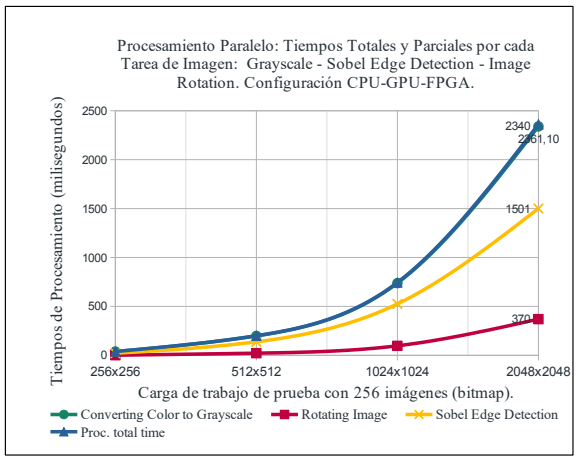
b)



c)



d)



e)

Legend. *PipeSkeleton:* Los datos mostrados en las columnas de la Tabla 5.4(a) presentan los resultados de medir los tiempos de procesamiento de cada dispositivo (sin considerar la comunicación ni entrada/salida), de comunicación, de ejecución (con y sin entrada/salida), de entrada/salida y los indices de aceleración (con y sin entrada/salida) de las pruebas de evaluación de rendimiento. En las Gráficas (b), (c), (d) y (e) se muestran los tiempos individuales de procesamiento de cada configuración y dispositivo (sin comunicación ni entrada/salida) de las imágenes por cada tarea u operador de imágenes con dimensiones 256, 512, 1024 y 2048 pixeles, en las diferentes configuraciones heterogéneas CPU, GPU y FPGA.

5.5.1.1 Sobre los tiempos de procesamiento:

En general, estos valores de la Tabla 5.4 (a) muestran principalmente los tiempos de procesamiento paralelo en CPU, GPU y FPGA ($Tpar_{cpu}, Tpar_{gpu}$ *y* $Tpar_{fpga}$) con **PipeSkeleton** donde se observa que son significativamente menores a los tiempos de procesamiento secuencial ($Tsec_{cpu}$).

Por ejemplo, se obervó una diferencia de tiempo de procesamiento mucho mas acentuada cuando el tamaño de los datos era grande. Por ejemplo en la configuración 0-GPU-0 el $Tsec_{cpu} >> Tpar_{gpu}$ (22632,67 miliseg. con respecto a 1512,34 miliseg. respectivamente en GPU) en el caso de la lista de 256 imágenes de 2048x2048 pixeles. Esto se mantuvo en las diferentes configuraciones de prueba con CPU, GPU y FPGA realizadas. Además, esta relación resultó que el $Tsec_{cpu}$ fué alrededor del orden de 20 veces superior, aunque cuando el tamaño de los datos eran pequeños (de 256x256 y 512x512 pixeles) la diferencia de estos tiempos se mantuvo levemente por debajo.

Así, como en el caso secuencial, se observó que los tiempos paralelos en cada configuración de computación de cada dispositivo se incrementaron cuando se aumentó el tamaño de los datos o resolución de las imágenes de entrada, cuya diferencia fué aprox. a dos ordenes de magnitud, aunque inferior a 100 veces.

Por ejemplo, con la configuración de sólo GPU (0-3-0), el tiempo de procesamiento en GPU ($Tpar_{gpu}$) con 256x256 piexeles fué de 49,90 milisegundos, mientras que con 2048x2048 pixeles fué de 4639,70 milisegundos. Un comportamiento similar se determinó en la configuración con CPU-GPU-FPGA, con 256x256 piexeles fué de 64,17 milisegundos, y con 2048x2048 pixeles de 6042,92 milisegundos. La misma diferencia de tiempo de procesamiento se presentó en la configuración de sólo FPGA ($Tpar_{fpga}$) , con 256x256 piexeles fué de 22,06 milisegundos, y con 2048x2048 pixeles de 1810,61 milisegundos, inferior a 85 veces.

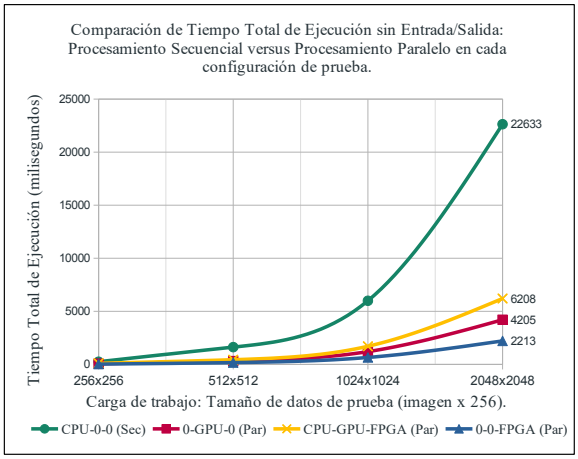
También, se observó que la tarea asociada al operador Sobel's Edge detection fué la tarea que consumió mayor tiempo de procesamiento en cada dispositivo de computación y configuración de prueba.

Asimismo, en las Gráficas adjuntas (b), (c), (d) y (e) se muestran las curvas de rendimiento extraidas de los datos de la Tabla 5.4. En estas gráficas se muestra un comportamiento no lineal o curva exponencial notablemente más pronunciada en el caso secuencial con relación a las curvas de procesamiento paralelo en las diferentes configuraciones con los dispsitivos GPU y FPGA.

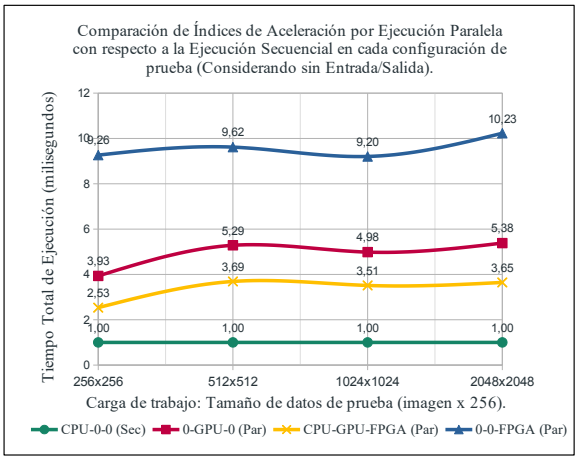
En la Gráfica (f) se observan los tiempos totales de procesamiento de las diferentes listas de imágenes según el tamaño de los datos procesados (resolución en piexeles). Se muestran las curvas por tiempo de procesamiento por volumen de datos en cada configuración heterogénea de prueba. Se observa que el tiempo de procesamiento es mucho mayor en la configuración CPU-0-0 sobre las demás configuraciones con GPU y FPGA, siendo muy pronunciada la diferencia con la lista de imagenes de 2048x204 pixeles con respecto al procesamiento de la lista de iamgenes con 256x256 pixeles, lo que reafirma lo observado en las gráficas (b), (c), (d) y (e).

Por otro lado, en la Gráfica (g) se muestran los tiempos desde la perspectiva del dispositivo de procesamiento. Aquí se notó mayor tiempo de cómputo realizado en la configuración con sólo CPU, y menor tiempo de cómputo con sólo FPGA. El tiempo de procesamiento con la configuración CPU-GPU-FPGA se observó siempre un rendimiento promedio mejor que el caso con CPU, pero menor a las configuraciones con GPU y FPGA.

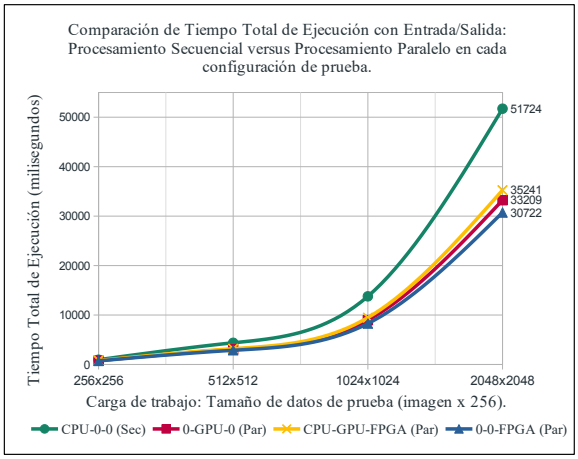
Cont. Tabla 5.4: Tiempos de procesamiento usando PipeSkeleton sobre CPU, GPU y FPGA.



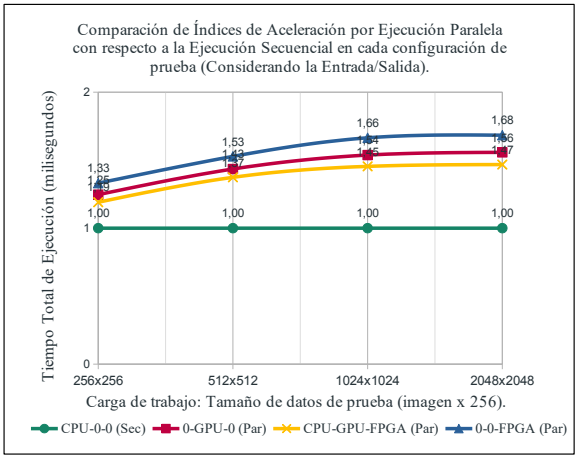
f)



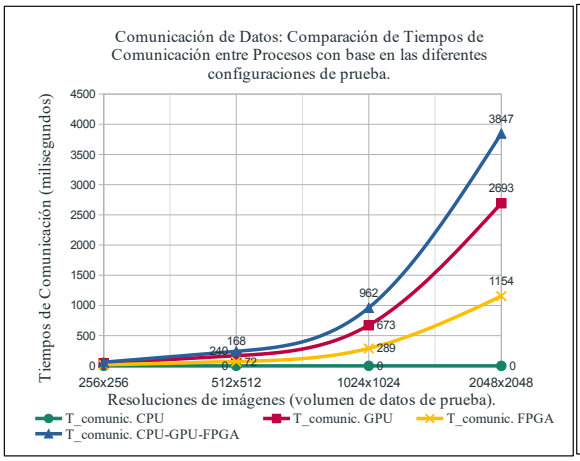
g)



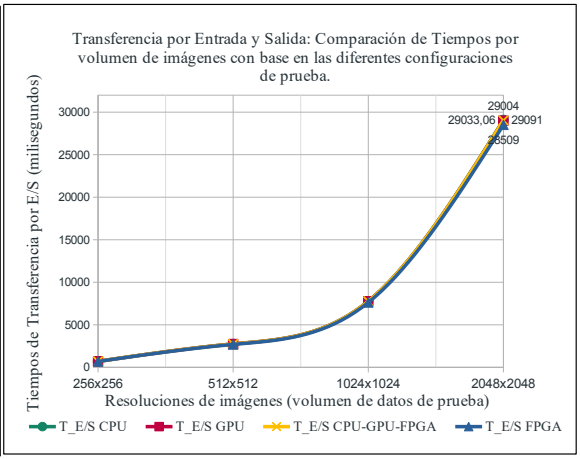
h)



i)



j)



k)

Legend (cont.). *PipeSkeleton:* Las Gráficas (f) y (h) muestran los tiempos de ejecución, sin y con entrada/salida, y las gráficas (g) e (i) los respectivos índices de aceleración respecto del tiempo secuencial, luego de procesar cuatro listas de 256 imágenes por configuración heterogénea. Las Gráficas (j) describe la curva de tiempos de comunicación entre procesos, y la gráfica (k) muestra los tiempos de transferencias de datos y resultados por entrada/salida.

5.5.1.2 Sobre los tiempos de comunicación y entrada/salida:

Los resultados registrados en las Gráficas (h) e (i) presentan los tiempos de comunicación y transferencia de datos por entrada/salida respectivamente, según cada configuración. En la Gráfica (i) se observó un considerable consumo de tiempo en el proceso de entrada de las listas de 256 imágenes y sus resultados de salida para cada resolución de imagen en cada dispositivo, mucho mayores a los tiempos de procesamiento y comunicación. Por otra parte, la otra Gráfica (h) con los tiempos de comunicación refleja el consumo de tiempo debido al intercambio de datos entre las tareas paralelas que se ejecutan en cada dispositivo de cómputo.

A diferencia de los tiempos de entrada y salida de datos, los tiempos de comunicación se muestran mucho menores, aunque siguen siendo mayores a los tiempos de procesamiento en cada configuración de prueba con **PipeSkeleton**.

También, se sigue observando en las gráficas que en cada configuración paralela hay un incremento levemente exponencial (no lineal) del tiempo de procesamiento paralelo y de comunicación en la medida que crece el tamaño de las imágenes de entrada.

5.5.2 *RESULTADOS CUALITATIVOS DE LA PRUEBA DE FUNCIONALIDAD Y ABSTRACCIÓN*

5.5.2.1 Sobre la funcionalidad y abstracción del esqueleto paralelo:

Para el experimento se codificaron dos programas fuente que implementan el programa paralelo de procesamiento de imágenes de prueba.

El código fuente mostrado en 5.12 fue desarrollado explotando de forma explícita el paralelismo usando el API OpenCL para el lenguaje C/C++, obteniendo un programa con una longitud de 6 páginas, con 60 líneas por página aproximadamente.

El otro código fuente que se observa en 5.14 fué desarrollado para explotar de forma implícita y estructurada paralelismo usando principalmente el esqueleto reconfigurable **PipeSkeleton**, además de algunas instrucciones en lenguaje C/C++ y del API OpenCL, pero mayoritariamente los componentes de la PAPI **SkeletonCoRe**. Con **PipeSkeleton** se ocultó el paralelismo en CPU, GPU y FPGA, como un mecanismo de encapsulado y abstracción que redujo la extensión del código fuente a una sola página, con casi 60 líneas aproximadamente. En el código fuente en 5.13 se muestra parte del cuerpo de código de **PipeSlkeleton** y el archivo con la cabecera o header ”skeletoncore.hpp“ de **PAPI SkeletonCoRe** empleado en dicho código se puede encontrar en el Anexo C, código fuente B.19.

Luego, de la ejecución de las dos implementaciones paralelas del programa de procesamiento de imágenes, explícita sin esqueletos e implícita con el esqueleto reconfigurable **PipeSkeleton**, se observó que ambas generaron las mismas imágenes resultados, mostrándondose dichos programas funcionalmente equivalentes, aunque no iguales en su codificación.

En la Figura 5.3 se muestra un ejemplo de las imágenes esperadas, producto del procesamiento digital de una imagen de la lista de imágenes originales a color procesadas. Como se mencionó anteriormente al comienzo de la sección 5.5, estas se han obtenido a partir de la conversión de las imágenes a color a imágenes en escala de grises, otras imágenes con detección de bordes de los objetos en la imagen usando el algoritmo de Sobel, y las últimas imágenes rotadas 90 grados hacia la derecha (en sentido del reloj).

Para culminar la evaluación de la funcionalidad de la **PAPI SkeletonCoRe**, en el siguiente **Capítulo 6** se realiza también las pruebas de rendimiento y de funcionalidad del esqueleto reconfigurable **TaskSkeleton**.

Se aplican los mismos criterios de comparación de los tiempos de ejecución, así como del tamaño y complejidad del código para resaltar el resultado esperado con respecto al ocultamiento y encapsulación del paralelismo a la vista del programador de la aplicación paralela. Asimismo, se resalta la capacidad de intercambiar, de forma fácil y transparente, las **particiones de cómputo** de software sobre el CPU y GPU y hardware sobre FPGA.

Capítulo 6

TaskSkeleton: Un Esqueleto Algorítmico Reconfigurable basado en el Modelo de Cómputo Paralelo *"Master-Slave"*

"El logro más impresionante de la industria del software es su continua anulación de los constantes y asombrosos logros de la industria del hardware."

Henry Petroski

6.1 INTRODUCCIÓN

Otro esqueleto al cual se le aplica una prueba de concepto para su evaluación es el esqueleto algorítmico reconfigurable denominado *TaskSkeleton* (Reconfigurable Tasks Skeleton). Este esqueleto algorítmico es otra instancia del catálogo de esqueletos reconfigurables de la interfaz de programación de aplicaciones paralelas (Parallel Applications Programming Interface) **SkeletonCoRe**.

6.2 ESPECIFICACIÓN Y DESCRIPCIÓN DEL ESQUELETO TASKSKELETON

Este esqueleto implementa el conocido patrón de procesamiento paralelo Maestro/Esclavo (Master/Slave) en el cual un "stream" de elementos de datos son distribuidos o repartidos por un proceso de control entre varios procesos esclavos a partir de éste proceso raíz o maestro. Aquí, cada proceso paralelo aplica una cadena de operaciónes o tareas a un flujo de datos de entrada.

Este comportamiento puede ser descrito así, $f\colon \alpha \rightarrow \beta$, donde α representa un "stream" de valores de entrada $a_0, a_1, \ldots, a_i, \ldots, a_{n-1}$ que son operados mediante la composición paralela de una función de control M que opera sobre $n-1$ arguementos o funciones subordinadas $s_1 \ldots s_{n-1}$ tal que $M(s_1(a_0) \rightarrow \gamma_1, \ldots, s_i(a_i) \rightarrow \gamma_i, \ldots, s_i(n_1) \rightarrow \gamma_{n-1}) \rightarrow \beta$. La idea es explotar el paralelismo a partir del procesamiento sobre diferentes elementos del "stream de entrada", siempre que no haya dependencias de datos.

Al igual que *PipeSkeleton*, el esqueleto *TaskSkeleton* también se diseña como una plantilla de alto nivel que se instancia con varios parámetros, entre datos y comportamientos, como el tamaño de las particiones de hardware y software, un arreglo de tareas para cada nodo esclavo del árbol Maestro/Esclavo, el tamaño del segmento de datos del "stream". Un ejemplo genérico del uso y parámetros del esqueleto es el siguiente:

TaskSkeleton*(CPUMaster, GPUSlave, FPGASlave, CPUtasks, GPUtasks, FPGAtasks, in_file, out_file)*,

donde: a) *CPUMaster, CPUSlave y GPUSlave:* son los componentes paralelos y particiones de cómputo del esqueleto Maestro/Esclavo de software en CPU y GPU, respectivamente; b) *y FPGASlave:* es la partición de cómputo de hardware en FPGA del Maestro/Esclavo, c) *CPUtasks, GPUtasks y FPGAtasks:* son listas de tareas de procesamiento asociadas a cada partición de cómputo del esqueleto Maestro/Esclavos; y d) *in_file y out_file:* son los archivos desde donde se lee y escribe el "stream de datos" de entrada y sus resultados.

El esqueleto *TaskSkeleton* gestiona y coordina internamente la interacción entre el Maestro con los Esclavos de procesamiento a través de sus particiones de cómputo, de software asignadas una al CPU y otra al GPU, y la otra de hardware asignada al FPGA. A cada partición le corresponden tareas y datos para procesamiento, incluyendo las tareas de entrada y salida. Para el programador es transparente la interfaz de comunicación que interconecta la partición de software en el CPU y GPU con la partición de hardware en la tarjeta FPGA. Internamente el esqueleto gestiona la creación de los procesos del Maestro y los Esclavos y el conjunto de asignaciones de datos desde el Maestro hacia los Esclavos y los resultados desde los Esclavos hacia el Maestro de la siguiente forma:

a) $CPUMaster \leftarrow \{CPUtasks, in\_file\}$

b) $CPUSlave \leftarrow \{CPUtasks, data\_segment\}$

c) $GPUSlave \leftarrow \{GPUtasks, data\_segment\}$

d) $FPGASlave \leftarrow \{FPGAtasks, data\_segment\}$

Est enfoque permite el codiseño integrado de componentes de hardware destinados al FPGA (como co-procesadores) y de componentes de software destinados al microprocesador CPU o GPU (como hilos o procesos). Así, es posible explorar espacios de diseño con el fin de intercambiar funcionalidades entre CPU-GPU y FPGA de forma transparente y elegir la combinación con mejor rendimiento.

6.3 DESCRIPCIÓN DEL EXPERIMENTO DE EVALUCIÓN DE TASKSKELETON

Como otra prueba de concepto se evalúa el costo y eficiencia del esqueleto paralelo *TaskSkeleton*. Para ello, se aprovecha que los programas poseen secciones o tareas que son intensivas en operaciones de entrada/salida de datos, y otras tareas intensivas en operaciones de cómputo o cálculo, a la que denominamos Kernels.

Al igual que la prueba de abstracción, funcionalidad y rendimiento del esqueleto paralelo anterior *PipeSkeleton*, aquí se aplica al esqueleto *TaskSkeleton* la misma aplicación de prueba de procesamiento digital de imágenes, donde se procesan varios operadores o kernels bajo la configuración de computación Maestro/Esclavo (árbol de un solo nivel) a una imagen. Por supuesto, acá no hay orden de precedencia de estos operadores de imagen, es variable:

a) CPU MasterTask → SlaveTask(Color to Grayscale Conversion, InputImage) → CPU MasterTask

b) CPU MasterTask → SlaveTask(Sobel's Edge Detection, InputImage) → CPU MasterTask

c) CPU MasterTask → SlaveTask(Rotating Image, InputImage) → CPU MasterTask

También, se usa la misma carga de trabajo de prueba, compuesta de cuatro listas de 256 imagenes c/u con diferentes dimensiones: 256x256, 512x512, 1024x1024 y 2048x2048 pixeles, respectivamente, para explotar paralelismo de datos y procesamiento intensivo.

En la Fig. 6.1 se puede ver la configuración de la aplicación de procesamiento de imágenes para la prueba de concepto, y en la Fig. 6.2 las configuraciones de computación heterogénea usadas para realizar la prueba de rendimiento y funcionalidad de **TaskSkeleton**.

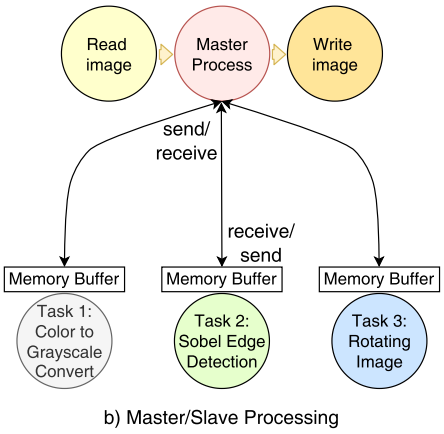


Figura 6.1: Procesamiento de imágenes con cómputo maestro/esclavo o master/slave. Fuente: Elaborado por el Autor.

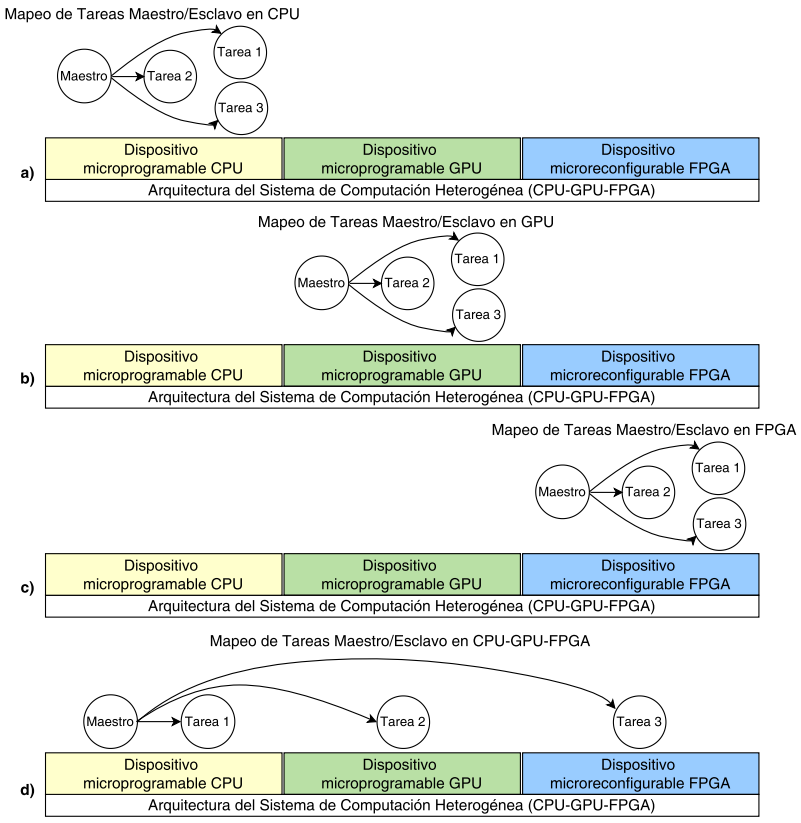


Figura 6.2: Configuración CPU-GPU-FPGA para la Ejecución de TaskSkeleton usando los Dispositivos OpenCL. Fuente: Elaborado por el Autor.

Para las configuraciones de prueba se han realizado particiones de las tareas de la forma CPU-GPU-FPGA. Inicialmente, como configuraciones iniciales y de referencia, se ha mapeado todo el árbol maestro/esclavo (tres etapas, un operador de imagen o kernel en cada dispositivo o etapa) (ver Fig. 6.2d), como tareas de software sólo en CPU (3-0-0, cpu-0-0) (ver Fig. 6.2a), luego sólo en GPU (0-3-0) (ver Fig. 6.2b), y por último se ha mapeado todo el pipeline como tareas de hardware sólo en el dispositivo FPGA (0-0-3) (ver Fig. 6.2c).

En todas las configuraciones de prueba, las tareas de lectura y escritura de datos se han mapeado de manera fija sólo en CPU del Host.

Por último, es importante mencionar que con propósitos de estudio, ésta aplicación de procesamiento de imágenes será la que se usará para la implementación, medición y comparación de las métricas de rendimiento de los esqueletos paralelos y el secuencial.

6.4 PROGRAMACIÓN DEL ESQUELETO RECONFIGURABLE TASKSKELETON

A continuación se implementa, compila y corre la aplicación paralela de prueba en OpenCL/C++e aplican tres operadores de imagen en árbol, usando el siguiente modelo:

$$CPUMaster \rightarrow CPUSlave(Color\ to\ Grayscale\ Conversion\ Task) \rightarrow CPUMaster.$$

$$CPUMaster \rightarrow GPUSlave(Sobel's\ Edge\ Detection\ Task) \rightarrow CPUMaster.$$

$$CPUMaster \rightarrow FPGASlave(Rotating\ Image\ Task) \rightarrow CPUMaster.$$

En esta aplicación paralela se aplica una cadena de tres operadores de imagen en pipeline como: → etapa 1 (Color to Grayscale Convert) → etapa 2 (Sobel's Edge Detection) → etapa 3 (Rotating Image) →, los cuales conforman una aplicación de procesamiento de imágenes digitales.

Acá se aprovecha el código fuente 5.12 del Capítulo 5 para mostrar una implementación sin esqueletos, es decir, programando de forma explícita el paralelismo sólo con instrucciones OpenCL C/C++. Luego, en el código fuente 6.15 se muestra la plantilla **TaskSkeleton** del catálogo de esqueletos reconfigurables de **SkeletonCoRe** y en el código fuente 6.16 se presenta el programa principal usado para codificar, compilar, correr y probar la misma aplicación paralela de procesamiento de imágenes digitales.

```
1  //*********************************************************************************************
2  //* Program name: TaskSkeleton of PAPI SkeletonCORE
3  //* Programmer  : Carlos Acosta-León
4  //* Function    : TasSkeleton Body.
5  //* Language/API: openCL C/C++
6  //*********************************************************************************************
7  //*** Definition of Tasks Function +++//
8  void Task1_convertImageGrayscale(unsigned char *inImage, unsigned char *outImage[], int width, int height,
9                int *channels);
10 void Task2_sobelEdgeDetection(unsigned char *inImage, unsigned char *(outImage[]), int iWidth, int iHeight,
11               int channels);
12 void Task3_imageRotated(unsigned char *inImage, unsigned char *(outImage[]), int width, int height,
13               int channels);
14 //*** Define an Array with pointer to functions: Pointer Array to Processing Functions
15 void (*tasksArray[MAXSIZE_TASKSLIST])(unsigned char *inImage, unsigned char *outImage[], int width,
16     int height, int *channels, double &exec_time) = {
17       *Task1_convertImageGrayscale(),
18       *Task2_sobelEdgeDetection(),
19       *Task3_imageRotated(),
```

Código fuente 6.15: **Programación Paralela Implícita:** Cuerpo del código en OpenCL/C++ del esqueleto **TaskSkeleton** para la aplicación paralela de procesamiento de imágenes de prueba. Fuente: Elaborado por el autor.

```
20    };
21  /*** PipeSkeleton Skeleton Body for Pipeline Parallel Processing ***/
22  void pipeSkeleton(skeletoncore::devicePartition partition1, skeletoncore::devicePartition partition1,
23            skeletoncore::devicePartition partition1, void (*tasksArray[])(void), int num_tasks) {
24      //* Data
25    workload_size = 256;
26    //* Create Communication Channels between devices CPU, GPU and FPGA
27    channel float cpu_gpu_channel, gpu_fpga_channel, fpga_cpu_channel;
28    //* Get the device(s)
29    err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &devices[1], &partition1.num_devices);
30    err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &devices[2], &partition2.num_devices);
31    err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_FPGA, 1, &devices[3], &partition3.num_devices);
32    //* Create contexts for devices CPU, GPU and FPGA
33    cl_context context;
34    context = clCreateContext(0, 3, devices, NULL, NULL, NULL, &err);
35    //* Create commands queue for devices
36    cl_command_queue queue_gpu, queue_cpu, queue_fpga;
37    partition1.device_cpu  = clCreateCommandQueue(context, devices[1], 0, &err);
38    partition2.device_gpu  = clCreateCommandQueue(context, devices[2], 0, &err);
39    partition3.device_fpga = clCreateCommandQueue(context, devices[3], 0, &err);
40    //* Basic pipeline for PieSkeleton
41    //*** Executing on CPU
42    __kernel void device_cpu(__global int* in) {
43      for (int i = 0; i < workload_size; ++i) {
44          int *image_in = read_ListImage();          // Get images by segments
45          image_out[partition1.dataSet.images.properties.width * partition1.dataSet.images.properties.height]
46            = partition1.kernel.(*tasksArray[1])(&image_in, 0, width*height, channelRGB); // do some work
47          write_channel(cpu_gpu_channel, &image_out);     // send data to the next partition
48        }
49    }
50    //*** Executing on GPU
51    __kernel void device_gpu() {
52      for (int i = 0; i < workload_size; ++i) {
53        int *image_in = read_channel(cpu_gpu_channel);  // take data from cpu
54        image_out[partition2.dataSet.images.properties.width * partition2.dataSet.images.properties.height]
55            = partition2.kernel.(*tasksArray[2])(&image_in, 0, width*height, channelRGB); // do some work
56        write_channel(gpu_fpga_channel, &image_out);     // send data to the next partition
57      }
58    }
59    //*** Executing on FPGA
60    __kernel void device_fpga(__global int* out) {
61      for (int i = 0; i < workload_size; ++i) {
62        int *image_in = read_channel(gpu_fpga_channel);          // take data from gpu
63        image_out[partition3.dataSet.images.properties.width * partition3.dataSet.images.properties.height]
64            = partition3.kernel.(*tasksArray[2])(&image_in, 0, width*height, channelRGB); // do some work
65        write_channel(fpga_cpu_channel, &image_out);             // write result in the end
66      }
67    }
68    //* Start concurrently tasks (kernels) for partitions in CPU-GPU-FPGA
69    clEnqueueTask(device_cpu, *partition1.kernel.(*tasksArray[1]));
70    clEnqueueTask(device_gpu, *partition2.kernel.(*tasksArray[2]));
71    clEnqueueTask(device_fpga,*partition3.kernel.(*tasksArray[3]));
72    clFinish(device_fpga); // last kernels in our pipeline
73  }
```

(Cont. Código fuente 6.15) **Programación Paralela Implícita:** Cuerpo del código en OpenCL/C++ del esqueleto **TaskSkeleton** para la aplicación paralela de procesamiento de imágenes de prueba. Fuente: Elaborado por el autor.

A continuación, en el código fuente 6.16, se puede ver un ejemplo donde se muestra un Programa Principal que usa el Esqueleto Reconfigurable PipeSkeleton y demás componentes de la **PAPI SkeletonCoRe** para explotar paralelismo de forma implícita sobre una plataforma de cómputo heterogéneo basada en CPU, GPU

y FPGA. Observe la poca complejidad y brevedad del código. Acá, se oculta la implementación paralela en OpenCL C/C++ de una aplicación de procesamiento de imágenes de prueba. El código fuente de la cabecera (header "skelcore.hpp") que muestra el detalle de la implementación de la **PAPI SkeletonCoRe** se puede encontrar en el Anexo C, código fuente B.19.

```
//*********************************************************************************************
//* Program name: Library of Objects and Parallel Skeletons of PAPI SkeletonCORE
//* Programmer  : Carlos A. Acosta-León
//* Function    : Show the TaskSkeletons Usage.
//* Language/API: OpenCL C/C++
//* Date        : 25/02/2022
//*********************************************************************************************
#include "papi-skeletoncore-header_ver_1-4.hpp"
//* Define User's Tasks or Functions for processing
#include "user_define_tasks_body_ver_1-4.hpp"
#define BILLION 1000000000L;
const   int MAXSIZE_TASKSLIST = 7;

// Declare space of names for SkeletonCoRe
using namespace skeletoncore_papi;
//using namespace user_tasks_body;
using namespace std;

/******* BEGIN MAIN PROGRAM ********************************************************/
int main(int argc, char *argv[]) {
    //* Declare data variables
    string  imageType     = "jpg"; // type of jpg, png, etc, for example
    string  mytasksPool[3];
    //* FIRST STEP: DECLARE INSTANCES OF SKELETONCORE OBJECTS
    //* Declare skeletoncore objects
    skeletoncore::papi_skeletoncore skelcore;  // Create the setup enviorement
    skeletoncore::dataSet           mydataSet; // Create container for images data/operations
    // device Partition for CPU, GPU and FPGA Processing
    skeletoncore::devicePartition   myCPUPartition, myGPUPartition, myFPGAPartition;
    //* SECOND STEP: SETUP THE SKELETONCORE ENVIORENMENT AND COMPUTING PARTITIONS
    //* Initialize the skeletoncore processing enviorement
    skelcore.init(argc);
    //* Getting info of computing devices in platform
    skelcore.platformDiscovery(skelcore.deviceGroup);
    //* Getting input data: Here it gests all properties from input image readed
    mydataSet.images.readDataSetImage("jpg", argv[1], mydataSet, &mydataSet.images.input_Images);
    //* Definition of Tasks body: Must be inside the "user's_define_tasks_body.hpp" header file
    skelcore.getTasksKernel(mytasksPool);
    //* THIRD STEP: SETUP THE COMPUTING PARTITIONS
    //* Setting Up Partitions for compute devices
    skelcore.partitionSetup(skelcore.deviceGroup, myCPUPartition, mydataSet.images.cpu,  mytasksPool[0]);
    skelcore.partitionSetup(skelcore.deviceGroup, myGPUPartition, mydataSet.images.gpu,  mytasksPool[1]);
    skelcore.partitionSetup(skelcore.deviceGroup, myFPGAPartition, mydataSet.images.fpga,mytasksPool[2]);
  //* FOURTH STEP: EXECUTE TASKSKELETON PARALLEL SKELETON FROM SKELETONCORE PAPI
    skelcore.exec.taskSkeleton(myCPUPartition, myGPUPartition, myFPGAPartition);
    //* FIFTH STEP: GET AND WRITE RESULTS AFTER PROCESSING
    //* Putting out results into Files
    mydataSet.images.writeOutputImages("image-out-list.jpg", mydataSet.images.properties.width,
      mydataSet.images.properties.height, mydataSet.images.properties.channels, mydataSet.images.out_);
    //* SIXTH STEP: CLOSING SKELETONCORE ENVIORENMENT
    skelcore.terminate();
    return 0; /*** END MAIN PROGRAM ***/
}
```

Código fuente 6.16: **Programación Paralela Implícita:** Código que muestra un Programa Principal que usa el Esqueleto Reconfigurable TaskSkeleton y demás componentes de la **PAPI SkeletonCoRe** para explotar paralelismo de forma implícita. El código fuente del header "papi-skeletoncore.hpp" se encuentra en el Anexo C, código fuente B.19. Fuente: Elaborado por el autor.

Al compilar y correr el programa en OpenCL/C++ de procesamiento de imagen con TaskSkeleton mostrado en el código fuente 6.16 obtenemos la siguiente salida:

```
$> g++ taskskeleton.cpp -o taskskeleton.out -lOpenCL
$> ./taskskeleton.out
===>  ******* Beginning of the SKELETONCORE PAPI Environment *******
--> 00.   Discovering the Opencl Platform and Devices...!
00.1  OpenCL Platforms Available on this Heterogeneous System: 2
00.2  Plataform ID=0 --> OpenCL Divice detected: Intel(R) Core(TM) i5-10400F CPU @ 2.90GHz
00.2  Plataform ID=1 --> OpenCL Divice detected: NVIDIA GeForce GTX 1060 3GB
00.2  Plataform ID=2 --> OpenCL Divice detected: Intel Stratix 10 FPGA
--> 01. ==>Leyendo Datos de la Carga de Trabajo a Procesar...!
--> 02. ==>Configurando la Partición de Cómputo del: CPU
02.1  Creando su contexto de procesamiento
02.2  Creando la cola de comandos del CPU
02.3  Creando objetos de memoria para datos
02.4  Copiando los datos al Dispositivo de Cómputo
03.1  Configurando el Kernel asociado a la Partición...
03.2  Creando y compilando el programa y Kernel OpenCL!
03.2  Extracción de Kernels de programa OpenCL
-->  2.   Leyendo Vectores de Datos de la Carga de Trabajo a Procesar...!
--> 02.   ==>Configurando la Partición de Cómputo del: GPU
02.1  Creando su contexto de procesamiento
02.2  Creando la cola de comandos del GPU
02.3  Creando objetos de memoria para datos
02.4  Copiando los datos al Dispositivo de Cómputo
03.1  Configurando el Kernel asociado a la Partición...
03.2  Creando y compilando el programa y Kernel OpenCL!
03.2  Extracción de Kernels de programa OpenCL
--> 02. ==>Configurando la Partición de Cómputo del: FPGA
02.1  Creando su contexto de procesamiento
02.2  Creando la cola de comandos del FPGA
02.3  Creando objetos de memoria para datos
02.4  Copiando los datos al Dispositivo de Cómputo
03.1  Configurando el Kernel asociado a la Partición...
03.2  Creando y compilando el programa y Kernel OpenCL!
03.2  Extracción de Kernels de programa OpenCL
--> 04.  Executing Parallel Skeleton on CPU
--> 05.   Copiando resultados de salida hacia el Host o Anfitrión!
--> 04.  Executing Parallel Skeleton on GPU
--> 05.   Copiando resultados de salida hacia el Host o Anfitrión!
--> 04.  Executing Parallel Skeleton on FPGA
--> 05.   Copiando resultados de salida hacia el Host o Anfitrión!
--> 06.  ===> TIEMPOS DE COMPUTACIÓN:
06.1  CPU  Time: 77458 ns
06.2  GPU  Time:  1312 ns
06.2  FPGA Time:   253 ns
07.2  ===> RESULTADOS GRABADOS EN: archivo_salida_cpu.dat
07.2  ===> RESULTADOS GRABADOS EN: archivo_salida_gpu.dat
07.2  ===> RESULTADOS GRABADOS EN: archivo_salida_fpga.dat
===>  ******* Ending of the SKELETONCORE PAPI Environment *******
```

6.5 EVALUACIÓN DE FUNCIONALIDAD Y RENDIMIENTO DE TASKSKELETON

Al igual que el caso de procesamiento paralelo usando el esqueleto **PipeSkeleton**, obtenemos el mismo resultado con el esqueleto **TaskSkeleton**, tal y como se observa en la Figura 6.3. Se produjeron las imagenes esperadas producto del procesamiento digital de la imagen original a color, donde en una se obtiene la conversión de la imagen a escala de grises, otra imagen con detección de bordes usando el algoritmo de Sobel, y la última imagen rotada 90 grados, es decir, rotada hacia la derecha en sentido del reloj.

a) Imagen de entrada original (1024x1024). b) Imagen original Convertida a Escala de Grises (1024x1024). c) Imagen con Detección de Bordes con Sobel (1024x1024). d) Imagen original rotada 90 grados (1024x1024).

Figura 6.3: Imagenes resultantes del procesamiento digital usando el esqueleto **TaskSkeleton**: a) Imagen original a color RGB (formato .jpg), b) Imagen convertida a escala de 256 grises, c) Imagen con bordes de objetos resaltados (con máscara Sobel), y d) Imagen con Rotación. Fuente: Elaborado por el autor.

6.5.1 *RESULTADOS DE LAS PRUEBAS DE RENDIMIENTO*

Los resultados de las pruebas de **TaskSkeleton** se muestran en la Tabla 6.5 (a) donde se presentan los tiempos de procesamiento paralelo de cuatro listas de 256 imágenes de dimensiones 256x256, 512x512, 1024x1024 y 2048x2048 pixeles, siendo cada imagen es una unidad individual de procesamiento. Se observan los tiempos de procesamiento cuando se implementa **TaskSkeleton** sólo en CPU (3-0-0). Estos tiempos se toman como referencia de tiempo secuencial en la determinación del índice de aceleración del cómputo con respecto a los tiempos del esqueleto **TaskSkeleton** configurado sólo en GPU (0-3-0), sólo en FPGA (0-0-3) y la combinación de particiones hardware/software entre CPU, GPU y FPGA (1-1-1).

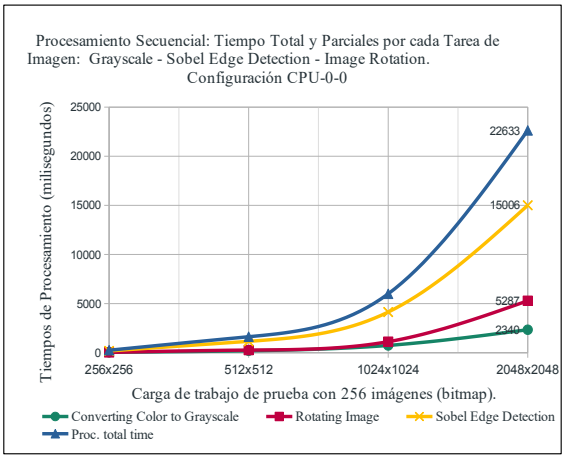
Asimismo, las Gráficas (b), (c), (d) y (e) adjuntas a la Tabla 6.5 muestran las curvas de comparación de los tiempos de procesamiento paralelo de **TaskSkeleton** en las diferentes configuraciones en CPU, GPU y FPGA. También, en la Gráfica (f) se observan los tiempos totales de procesamiento de las diferentes listas de imágenes según su resolución (tamaño de los datos). Se muestran las curvas por tiempo de procesamiento por volumen de datos en cada configuración heterogénea de prueba. Por otro lado, en la Gráfica (g) se muestran los tiempos de procesamiento desde la perspectiva del dispositivo de procesamiento. Siguen las Gráficas (h) e (i) donde se presentan los tiempos de transferencia de datos por entrada/salida y comunicación según cada configuración.

Tabla 6.5: Tiempos de procesamiento usando TaskSkeleton sobre CPU, GPU y FPGA.

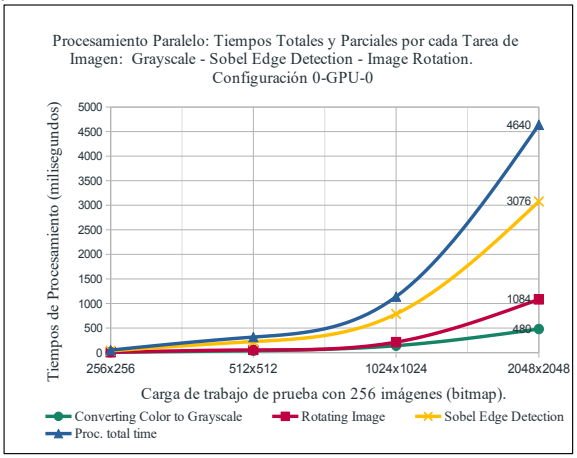
OPENCL, USANDO ESQUELETOS ALGORÍTMICOS: TASKSKELETON

Configuración CPU-GPU-FPGA	Carga de trabajo (Workload)		Tarea 1: Color to Grayscale Convert	Tarea 2: Sobel Edge detection	Tarea 3: Color Image Rotation	Tiempo Total de Proc.	Tiempo de Comunicac. entre Procesos	Tiempo lectura archivos (Entrada)	Tiempo escritura archivos (Salida)	Tiempo Total de Entrada/ Salida	Tiempo Total de Ejecución	Aceleración (SpeedUp)
Partición de Cómputo	Cantidad de Imagenes (Carga de Trabajo)	Dimensión NxN o Tamaño de Imagen (pixels)	Tiempo proc. Color to Grayscale x 256 imágenes (miliseg)	Tiempo proc. Sobel Edge Detection x 256 imágenes (miliseg)	Tiempo proc. Image Rotation x 256 imágenes (miliseg)	Tiempo CPU-GPU-FPGA x 256 imágenes (milisegs.)	Tiempo de Comunic. CPU-GPU-FPGA (milisegs.)	Tiempo de lectura x 256 imágenes (miliseg.)	Tiempo de grabar x 256 imágenes (milisegs.)	Tiempo Total Comunicación x 256 imágenes (miliseg)	Tiempo Proc. + Tiempo Comunic. + Tiempo E/S(miliseg)	Tsec/Tpar
3-0-0, CPU Caso Secuencial en C/C++, sin OpenCL	256	256x256	38,53	179,10	33,05	250,68	0,00	197,40	508,12	705,52	956,20	1,00
		512x512	198,54	1159,01	268,46	1626,01	0,00	835,19	1915,41	2750,59	4376,61	1,00
		1024x1024	738,08	4137,25	1119,17	5994,49	0,00	2631,02	5148,43	7779,46	13773,95	1,00
		2048x2048	2339,54	15006,18	5286,94	22632,67	0,00	7365,16	21726,08	29091,24	51723,91	1,00
0-3-0, GPU	256	256x256	5,54	25,75	4,59	35,88	33,75	197,40	508,12	360,16	429,79	6,99
		512x512	27,55	160,83	38,60	226,99	135,01	835,19	1915,41	1404,15	1766,15	7,16
		1024x1024	100,83	565,19	152,89	818,91	540,03	2631,02	5148,43	3971,34	5330,28	7,32
		2048x2048	344,84	2211,84	779,27	3335,94	2160,14	7365,16	21726,08	14850,79	20346,87	6,78
1-1-1, CPU-GPU-FPGA	256	256x256	7,09	32,96	6,08	46,14	40,41	197,40	508,12	431,18	517,73	5,43
		512x512	32,97	192,50	44,59	270,06	161,63	835,19	1915,41	1681,03	2112,72	6,02
		1024x1024	120,46	675,25	182,66	978,38	646,52	2631,02	5148,43	4754,42	6379,31	6,13
		2048x2048	449,13	2880,78	1014,95	4344,86	2586,08	7365,16	21726,08	17779,11	24710,05	5,21
0-0-3, FPGA	256	256x256	2,44	11,33	2,09	15,86	16,64	197,40	508,12	177,54	210,04	15,80
		512x512	12,13	70,83	16,41	99,37	66,55	835,19	1915,41	692,19	858,11	16,36
		1024x1024	42,98	240,95	65,18	349,11	266,21	2631,02	5148,43	1957,70	2573,03	17,17
		2048x2048	134,57	863,16	304,10	1301,83	1064,86	7365,16	21726,08	7320,81	9687,50	17,39

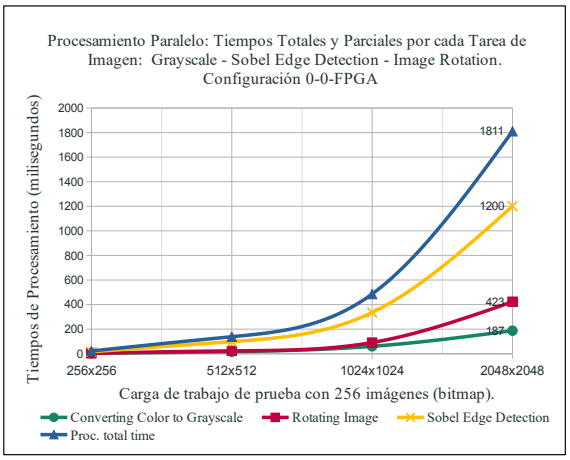
a)



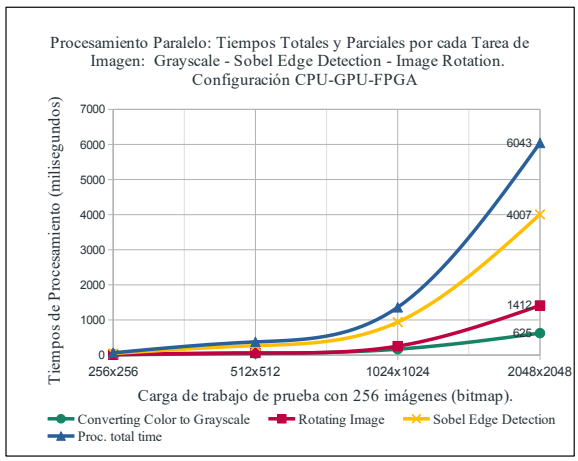
b)



c)



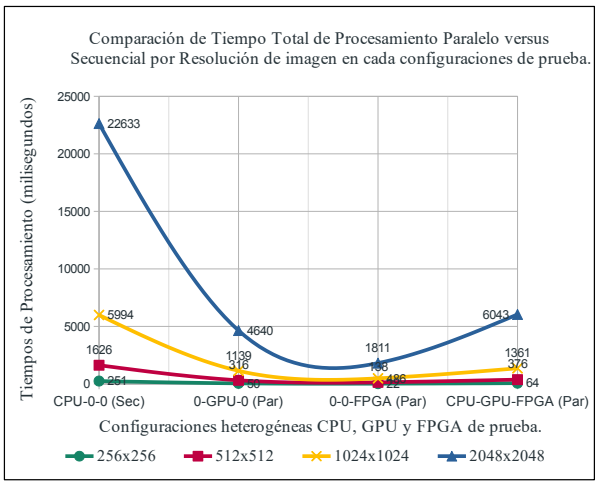
d)



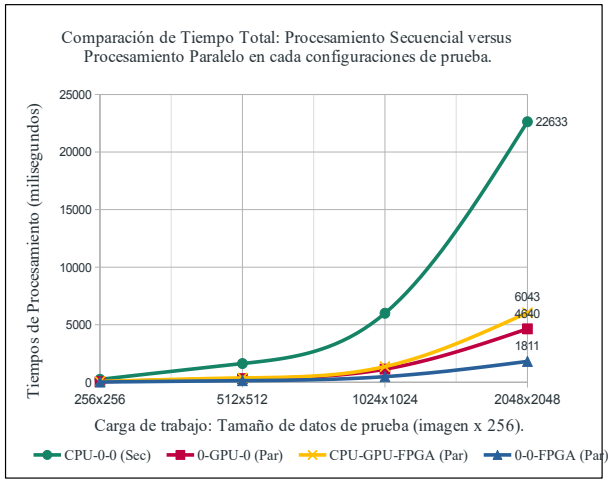
e)

Legend: *TaskSkeleton:* La Tabla mostrada en (a) presenta los datos resultantes de medir los tiempos de ejecución de las pruebas de evaluación de rendimiento. En las Gráficas (b), (c), (d) y (e) se muestran los tiempos individuales de procesamiento de las imagenes por cada operador de imagenes con dimensiones 256, 512, 1024 y 2048 pixeles, en las diferentes configuraciones heterogéneas CPU, GPU y FPGA.

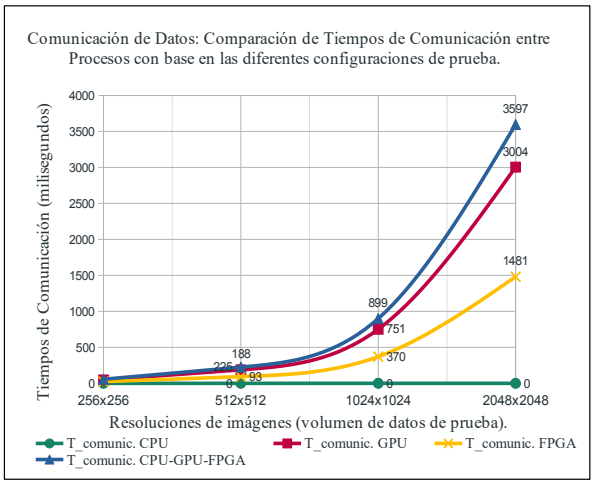
Cont. Tabla 6.5: Tiempos de procesamiento usando TaskSkeleton sobre CPU, GPU y FPGA.



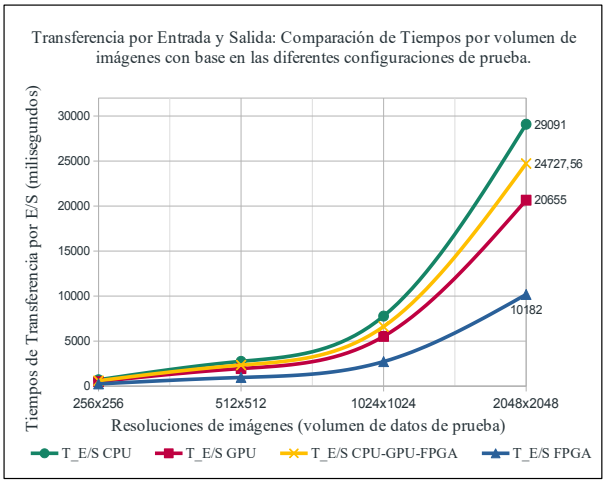
f)



g)



h)



i)

Legend: *TaskSkeleton:* La Gráfica (f) muestra el tiempo total de procesamiento de las listas de 256 imagenes según sus dimensiones en cada configuración heterogénea. La Gráfica (g) presenta los resultados de medir el tiempo de procesamiento por cada dispositivo de computación. Las Gráficas (h) e (i) concentran las curvas que describen el comportamiento de las transferencias de datos y resultados por entrada/salida y comunicación.

6.5.2 *RESULTADOS DE LAS PRUEBAS DE FUNCIONALIDAD Y ABSTRACCIÓN*

Finalmente, en las evaluaciones realizadas se puede observar que se mantiene la poca complejidad y brevedad del código fuente mostrado en 6.16. Aquí, en una sola página se expresa la aplicación de procesamiento de imágenes de prueba usando paralelismo implícito, donde su implementación en OpenCL C/C++ se encapsula en el esqueleto reconfigurable. Es decir, en éste código la solución oculta los detalles de implementación del paralelismo, diseño lógico, de la arquitectura heterogénea y naturaleza del dispositivo de cómputo. El código fuente de la cabecera (header "skeletoncore.hpp") que muestra el detalle de la implementación de la **PAPI SkeletonCoRe** se puede encontrar en el Anexo C, código fuente B.19.

Por otro lado, como comparación se observa también la evidente extension y mayor complejidad del programa OpenCL C/C++ puro sin esqueletos reconfigurables (donde en 6 páginas se explota el paralelismo de forma explícita con OpenCL C/C++) mostrado en el código fuente 5.12. Este enfoque explícito obliga

al programador a conocer y gestionar muchos detalles de la explotación explícita del paralelismo con OpenCL C/C++ en cada dispositivo de computación de la plataforma de cómputo heterogéneo, tanto los microprogramables como el CPU y GPU, como en los microreconfigurables, como el FPGA.

Adicionalmente, dado el nivel de abstracción logrado por la aplicación de las propiedades de la programación basada en objetos se obtuvo una interfaz simple y homogénea mostrando los esqueletos como plantillas. Esto es útil dado que resalta la capacidad de intercambiar, de forma fácil y transparente, las **particiones de cómputo** entre software (CPU y GPU) y hardware (FPGA) para hacer posible la exploración de diferentes espacios o configuraciones de diseño y rendimiento.

Culminada la evaluación de la funcionalidad de los esqueletos reconfigurables como parte inicial del portafolio de la **PAPI SkeletonCoRe**, en el próximo **capítulo 7** se procede a interpretar, comentar y discutir los resultados obtenidos en cuanto a la evaluación de rendimiento y funcionalidad de los esqueletos **PipeSkeleton y TaskSkeleton.**

Capítulo 7:

Discusión de resultados

"Lo que se puede afirmar sin evidencia, puede ser descartado sin evidencia."

Christopher Hitchens

7.1 DISCUSIÓN DE LOS RESULTADOS DE LA EVALUACIÓN DE LOS ESQUELETOS

A continuación se presentan algunas interpretaciones de los resultados obtenidos de las pruebas y experimentos aplicados a los esqueletos reconfigurables de la Interfaz de Programación de Aplicaciones Paralelas **SkeletonCoRe**.

Uno de los aspectos que se desprende de los resultados es la relación existente entre el incremento del tiempo de procesamiento de los dispositivos de cómputo en CPU ($Tproc_{cpu}$), GPU ($Tproc_{gpu}$) y FPGA ($Tproc_{fpga}$); y el aumento del tamaño de los datos. Este comportamiento es explicado en la teoría del paralelismo. En este sentido es de esperarse además, que dependiendo del tipo de algoritmo, se puede presentar que cuando el tamaño de los datos es pequeño o el algoritmo de cómputo es simple, la solución secuencial puede llegar a ser mejor que el tiempo de procesamiento de la solución paralela. Esto significa que hay que tener en cuenta aspectos del paralelismo como la creación, sincronizacion, comunicación y destrucción de procesos que representan un consumo de tiempo adicional (overhead time, delay time) debido a su gestión que afectan el tiempo total de procesamiento. Debido a que estos elementos no estan presentes en la solución secuencial, esto se presenta como una ventaja en la solución secuencial que permite obtener mejor tiempo con respecto a la solución paralela.

Otro aspecto que se pudo constatar es que aún los problemas con alta intensidad de datos o de computo, generalmente el paralelismo es una opción de mayor rendimiento, más aún cuando este se puede explotar como un algoritmo de baja granularidad en software en GPU y/o en hardware del FPGA.

7.1.1 *CON RESPECTO A LAS PRUEBAS DE RENDIMIENTO*

Por ejemplo, los resultados obtenidos de **PipeSkeleton**, mostrados en la Tabla 5.4 y sus Gráficas de curvas adjuntas, muestran un notable incremento de la aceleración del tiempo de ejecución (Speed-Up) en las pruebas de referencias de GPU alrededor de más del 50 %, y de FPGA cercano al 90 %, con respecto

a los tiempos de ejecución de PipeSkeleton corriendo sólo en CPU. Asimismo, se observa que las pruebas intermedias con combinaciones de particiones de PipeSkeleton con menos etapas en GPU y mas etapas en FPGA muestran un incremento progresivo y significativo del Speed-Up entre el 51.42 % y el 87.13 % con respecto a los tiempos de CPU. Estas pruebas demuestran también, que es posible explorar de forma flexible diferentes combinaciones de espacios de configuración o diseño de la aplicacion paralela para elegir la configuracion con mejor rendimiento y eficiencia.

Por otro lado, un comportamiento similar se obtuvo con los resultados de **TaskSkeleton**, observados en la Tabla 6.5 y en los Gráficos asociados a dicha Tabla. En ésta se muestran igualmente un importante incremento del Speed-Up en las pruebas de referencias de GPU hasta un 61 %, y de FPGA hasta un 77 %, con respecto a los tiempos de ejecución corriendo sólo en CPU. Asimismo, se observa que las pruebas intermedias con combinaciones de particiones sólo en GPU y sólo en FPGA muestran un incremento progresivo y significativo del Speed-Up entre el 47 % y el 81 % aprox. con respecto a los tiempos secuenciales de CPU.

Estas pruebas también demuestran que se mantiene la opción de explorar espacios de diseño de la aplicacion paralela para elegir la configuracion con mejor rendimiento y eficiencia.

Otra observación importante, obtenida de las Tablas y Gráficos, se refiere al hecho que la etapa del pipeline con mayor latencia constituyó un cuello de botella apreciable en el procesamiento en CPU y GPU, pero muy poco en FPGA.

También, consideramos que la cantidad de núcleos del GPU utilizado afecta el tiempo de procesamiento de la lista de imagenes cuando sus tamaños son mayores al tamaño del arreglo de núcleos del GPU, lo cual reduce el Speed-Up.

Finalmente, debido a que el CPU y, las tarjetas del GPU y FPGA, están conectados directamente al sistema de computacion heterogénea mediante un bus jerárquico de alta velocidad de 64 bits se observó una baja latencia de comunicación en la transferencia de la lista de 256 imagenes usadas en cada prueba.

7.1.2 *CON RESPECTO A LAS PRUEBAS DE FUNCIONALIDAD Y ABSTRACCIÓN*

Las implementaciones de los esqueletos algorítmicos **PipeSkeleton** y **TaskSkeleton** muestran evidentemente una menor complejidad y brevedad como se observa en el código mostrado en 5.14 donde en una sola página se explota el paralelismo de forma implícita, esto lo hace simple. En cambio, comparando la evidente extension del programa OpenCL C/C++ en 5.12 donde con 6 páginas se explota el paralelismo de forma explícita, esto lo hace complejo.

Esto se debe a que **PipeSkeleton** y **TaskSkeleton** encapsulan y ocultan la implementación paralela en OpenCL C/C++ de la aplicación de procesamiento de imágenes de prueba. El código fuente de la cabecera (header ”skeletoncore.hpp“) que muestra el detalle de la implementación de la **PAPI SkeletonCoRe** se puede encontrar en el Anexo C, código fuente B.19.

Asímismo, en el código fuente mostrado en 5.12 del capítulo 5, se puede ver la extensión y complejidad del código fuente en OpenCL C/C++ que implementa una aplicación de procesamiento de imágenes con sólo un operador usando de ejemplo el operador Sobel Edge detection. En este enfoque explícito el programador se ve obligado a conocer y gestionar muchos detalles explícitos para explotar el paralelismo, dado que es necesario considerar la interacción entre cada dispositivo de computación de la plataforma de cómputo heterogéneo, tanto los microprogramables como el CPU y GPU, como en los microreconfigurables, como el FPGA.

En contraste, en los códigos fuente de **PipeSkeleton** en 5.14 y de **TaskSkeleton** en 6.16 de los capítulos 5 y 6, se observa una reducción siginicativa de la extensión y complejidad del código fuente, además que la

expresión de la solución oculta los detalles del paralelismo y diseño lógico del FPGA, según la arquitectura o naturaleza del dispositivo de cómputo.

La evaluación de funcionalidad del esqueleto reconfigurable **TaskSkeleton** tuvo el mismo comportamiento que **PipeSkeleton**, es decir, lograron ocultar y encapsular el paralelismo a la vista del programador de la aplicación paralela. Asimismo, el uso de los esqueletos permitió también intercambiar de forma fácil y transparente las **particiones de cómputo** entre software sobre el CPU y GPU y hardware sobre FPGA. Esto se observa y evidencia en la comparación del tamaño y complejidad de los códigos fuente usando sólo el API OpenCL C/C++ en contraste con el código expresado con la **PAPI SkeletonCoRe**.

Como se ha observado, aunque el código desarrollado con el PAPI SkeletonCoRe en OpenCL/C++ pueda ejecutarse en varios dispositivos de cómputo, el rendimiento obtenido en cada dispositivo puede ser diferente. Es decir, unas aplicaciones tienen mejor desempeño que otras en un mismo dispositivo. tal como se muestra en la comparación de tiempos de ejecucuón del Filtro Sobel para Detección de Borde de imagenes en CPU, GPU y FPGA.

Puesto que el rendimiento generalmente depende en gran parte de cómo se adapta la solución a cada dispositivo de cómputo de la arquitectura heterogénea, crear aplicaciones optimizadas implica conocer los detalles de cada arquitectura. Es por ello, que se recomienda asignar las tareas con kernels que consumen mayor tiempo de cómputo al FPGA en primer lugar, o al GPU en segundo lugar.

Por último, dado el nivel de abstracción logrado por la aplicación de las propiedades de la programación basada en objetos se obtuvo una interfaz simple y homogénea mostrando los esqueletos como plantillas. Esto es útil dado que resalta la simplicidad de uso y la capacidad de intercambiar, de forma fácil y transparente, las **particiones de cómputo** entre software (CPU y GPU) y hardware (FPGA) lo cual hace posible la exploración de diferentes espacios o configuraciones de diseño y rendimiento.

Capítulo 8

Conclusiones y Trabajo Futuro

"*Lo que se puede afirmar sin evidencia, puede ser descartado sin evidencia.*"

Christopher Hitchens

8.1 CONCLUSIONES

Con base en los objetivos propuestos y alcanzados en la presente tesis, se presentan algunas conclusiones y futuras líneas de investigación que se derivan del presente trabajo de investigación doctoral.

Como aporte principal de la tesis se lograron desarrollar los componentes básicos del API **SkeletonCoRe**, una Interfaz de Programación de Aplicaciones Paralelas para Computación Heterogénea Reconfigurable.

Con respecto al primer objetivo del trabajo, se hizo una revisión conceptual y un estudio específico del enfoque de esqueletos algoritmicos, los detalles asociados al paralelismo y el diseño de hardware con FPGA. También, se revisaron herramientas como el método de codiseño de aplicaciones embebidas, el enfoque de funiones de orden superior y la programación basada en objetos.

Estos elementos fueron integrados en el API **SkeletonCoRe**, una herramienta de codiseño y programación de aplicaciones paralelas embebidas en sistemas heterogéneos reconfigurables del tipo CPU-GPU-FPGA.

En este sentido, el enfoque de esqueletos algorítmicos resultó un concepto útl para alcanzar la abstracción y ocultamiento suficiente de los detalles asociados a la implementación de las funcionalidades en CPU y GPU (tareas en software) y las del FPGA (tareas en hardware). Todo esto como parte del codiseño y programación de la aplicación paralela heterogénea.

Por tanto, la herramienta permite un enfoque de programación fácil para el desarrollo de prototipos de aplicaciones paralelas embebidas con tareas en un microprocesador o microcontrolador, interactuando con tareas de control que residen en un dispositivo de hardware como un FPGA, y que luego se pueden fijar en un chip ASIC para una solución computacional específica.

Con respecto al segundo objetivo, en **SkeletonCoRe** se diseñaron dos esqueletos algorítmicos paralelos y reconfigurables, uno para el algorítmo procesamiento paralelo denominado segmentación encauzada o pipeline

(**PipeSkeleton**), y el otro para el algoritmo denominado maestro esclavo o master/slave (**TaskSkeleton**), usando modelos de clases/objetos para ocultar sus estructuras y comportamientos.

Para cumplir con el tercer objetivo del trabajo, se programaron en OpenCL/C++ los esqueletos algorítmicos reconfigurables mencionados anteriormente. Estos se implementaron como funciones de orden superior usando el modelo de funciones del lenguaje C++ para construir plantillas que reciben como parámetros datos y tareas. Con esto se aprovechó del lenguaje C++ la capacidad de reutilización del código, la modularidad y el polimorfismo como elementos claves para lograr abstracción, encapsulamiento, movilidad de código y versatilidad de la herramienta y en consecuencia la estructuración del paralelismo.

Como último objetivo se hizo una evaluación de la funcionalidad y rendimiento de los Esqueletos implementados con OpenCL/C++. Para ello, y con el propósito de probar los conceptos y técnicas involucradas e integradas en **SkeletonCoRe**, se aplicaron pruebas de uso, utilidad y rendimiento a los esqueletos **PipeSkeleton** y **TaskSkeleton**. En consecuencia, se usaron como prueba de concepto algoritmos de procesamiento de imágenes intensivos en cómputo de datos corriendo en diferentes configuraciones heterogéneas del tipo CPU-GPU y FPGA.

Esta evaluación mostró que los esqueletos paralelos implementados producen los mismos resultados cualitativos (morfología y apariencia de las imágenes procesadas) y además mejoran sustancialmente los resultados cuantitativos de rendimiento en comparación con el mismo programa paralelo explícito en OpenCL/C++.

En fin, con las pruebas de rendimiento y funcionalidad de los Esqueletos Reconfigurables implementados en OpenCL/C++ se pudo explotar paralelismo implícitamente debido al nivel de abstracción y transparencia logrado con OpenCL/C++ y el enfoque de Esqueletos Algorítmicos.

Como valor agregado el método de codiseño, gracias al enfoque de esqueletos algorímicos, permitió de forma transparente intercambiar o migrar tareas entre software (CPU y GPU) y hardware (FPGA), para explorar espacios de diseño sobre el sistemas de computación heterogénea reconfigurable usado. También, permite automatizar las optimizaciones independientemente de la plataforma de computacion heterogénea que se dispone.

Con la arquitectura en capas del API de **SkeletonCoRe**, el programador tiene acceso directo tanto a los recursos e instrucciones nativas del lenguaje C++, al API de OpenCL/C++, y a la capa de Esqueletos Algoritmicos Reconfigurables, los cuales pueden combinarse de forma compatible en el cuerpo de un programa paralelo.

Otra ventaja es que los programas códificados usando los Esqueletos Reconfigurables de **SkeletonCoRe** son más compactos y legibles que aquellos programas codificados en OpenCL/C++ puro que expresan el paralelismo de forma explícita. Con esto el programador se abstrae de la implementación de una aplicación paralela dado que la herramienta le presenta una interfaz secuencial de alto nivel que permite explotar paralelismo implícito, lo cual reduce considerablemente la complejidad de programación.

Sin embargo, es necesario tener en cuenta que al margen del nivel de abstracción paralela que provee la herramienta, el programador no escapa del conocimiento de ciertos aspectos relacionados al problema, como es el caso de elegir el patrón de cómpúto paralelo más apropiado, o determinar qué tareas del algoritmo pueden procesarse en paralelo, o decidir qué tipo de paralelismo usar, de datos o tareas; la granularidad de éste, etc. Asímismo, el conocimiento de las características tecnológicas del dispositivo de computación puede ayudar a aprovecharlas para lograr una mejor optimización de la solución.

8.2 TRABAJO FUTURO

El trabajo de tesis plantea a futuro emprender más investigaciones similares o fortalecer la investigación realizada. Además, estas investigaciones pueden considerar el abordar la mejora de aquellos aspectos asociados al rendimiento y versatilidad de **SkeletonCoRe** que se presentaron en la tesis.

Un trabajo adicional es ampliar la librería de esqueletos que implementen otros patrones de cómputo paralelo como map & reduce, divide & conquer, tasks farm, data farm, etc. Además, se deben migrar los esqueletos de **SkeletonCoRe** a un lenguaje que permita su verificación funcional, como por ejemplo el lenguaje SystemC, el cual puede permitir mayor portabilidad y abstracción.

Asimismo, se deben hacer pruebas con aplicaciones más exigentes en poder de computación, como aquellas del área de la criptografia, de simulación numérica, inteligencia artificial (TensorFlow) por ejemplo; para evaluar la funcionalidad y medir el estres del rendimiento de los esqueletos sobre una paltaforma heterogénea particular.

Finalmente, se deben realizar evaluaciones para aumentar la rapidez de ejecución de la aplicación paralela por la via de aprovechar y explotar las caracteristicas tecnológicas de cada dispositivo para obtener una mejor optimización, la minimización de la sobrecarga de comunicación, la portabilidad de las aplicaciones, etc., pero principalmente la reducción del consumo de energía en los sistemas de computación heterogéneos CPU-GPU-FPGA.

Referencias bibliográficas

[Acar and Blelloch, 2019] Acar, U. A. and Blelloch, G. E. (2019). *Algorithms - Parallel and Sequential*. www.parallel-algorithms-book.com.

[Acosta-León and Cole, 2008] Acosta-León, C. and Cole, M. (2008). Fpga parallel programming based on skeletons: Image processing using algorithms into hardware. *Proceedings of the Conference on Parallel Computing Models and Applications, PCMA 2008, April 21-25, Glasgow, UK. 2008.*, 1:115–121.

[Acosta-León and Suros, 2022] Acosta-León, C. and Suros, R. (2022). Reconfigurable pipeline skeleton for parallel applications codesign on high-performance heterogeneous computing systems. *Proceedings of the Advanced Computing Trends Workshops, 9th Latin America High-Performance Computing Conference, CARLA 2022, Porto Alegre, Brazil, September 26–30, 2022.*, pages 23–30. Doi: 10.5281/zenodo.7469164.

[Babu and Parthasarathy, 2020] Babu, P. and Parthasarathy, E. (2020). Reconfigurable FPGA Architectures: A Survey and Applications. *Journal of The Institution of Engineers (India): Series B*, 102(1):143–156.

[Barkalov et al., 2019] Barkalov, A., Titarenko, L., and Mazurkiewicz, M. (2019). *Foundations of Embedded Systems*. Studies in Systems, Decision and Control 195. Springer International Publishing, 1st edition.

[Bell et al., 2018] Bell, S., Pu, J., and Hegarty, J. (2018). *Compiling Algorithms for Heterogeneous Systems*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 1st edition.

[Benkrid and Crookes, 2004] Benkrid, K. and Crookes, D. (2004). From Application Descriptions to Hardware in seconds: A logic-based Approach to Bridging the Gap. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(4):420–436.

[Benkrid, 2014] Benkrid, Khaled;Vanderbauwhede, W. (2014). *High-Performance computing using FPGAs*. Springer-Verlag New York, 2nd edition.

[Benoit et al., 2005] Benoit, A., Cole, M., Gilmore, S., and Hillston, J. (2005). Flexible skeletal programming with eskel. In Cunha, J. C. and Medeiros, P. D., editors, *Euro-Par 2005 Parallel Processing*, pages 761–770, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Bernstein, 1966] Bernstein, A. J. (1966). Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15:757–763.

[Bobda, 2007] Bobda, C. (2007). *Introduction to Reconfigurable Computing: Architectures, Algorithms and Applications*. Springer Dordrecht, 1st edition.

[Bobda et al., 2022] Bobda, C., Mbongue, J. M., Chow, P., Ewais, M., Tarafdar, N., Vega, J. C., Eguro, K., Koch, D., Handagala, S., Leeser, M., Herbordt, M., Shahzad, H., Hofste, P., Ringlein, B., Szefer, J., Sanaullah, A., and Tessier, R. (2022). The future of fpga acceleration in datacenters and the cloud. *ACM Trans. Reconfigurable Technol. Syst.*, 15(3).

[Booch et al., 2007] Booch, G., Maksimchuk, R. A., Engel, M. W., Young, B. J., Conallen, J., and Houston, K. A. (2007). *Object-oriented analysis and design with applications*. The Addison-Wesley object technology series. Addison-Wesley, 3rd edition.

[Caarls et al., 2006] Caarls, W., Jonker, P., and Corporaal, H. (2006). Skeletons and asynchronous rpc for embedded data and task parallel image processing. *IEICE - Trans. Inf. Syst.*, E89-D(7):2036–2043.

[Cardoso and (auth.), 2011] Cardoso, J. M. P. and (auth.), M. H. (2011). *Reconfigurable Computing: From FPGAs to Hardware/Software Codesign*. Springer-Verlag New York, 1st edition.

[Churiwala, 2017] Churiwala, S. (2017). *Designing with XilinxÂ® FPGAs: Using Vivado*. Springer International Publishing, 1st edition.

[Cole, 1989a] Cole, M. (1989a). *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press & Pitman, 1st edition.

[Cole, 2004a] Cole, M. (2004a). Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30.

[Cole, 2004b] Cole, M. (2004b). Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30.

[Cole, 1989b] Cole, M. I. (1989b). *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press & Pitman, 1st edition.

[Czarnul, 2018] Czarnul, P. (2018). *Parallel programming for modern high performance computing systems*. Chapman & Hall/CRC, 1st edition.

[David A. Patterson, 2020] David A. Patterson, J. L. H. (2020). *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, 2 edition.

[de Schryver et al., 2013] de Schryver, C., Marxen, H., Weithoffer, S., (auth.), N. W., Vanderbauwhede, W., and (eds.), K. B. (2013). *High-Performance Computing Using FPGAs*. Springer-Verlag New York, 1st edition.

[Dubey, 2009] Dubey, R. (2009). *Introduction to Embedded System Design Using Field Programmable Gate Arrays*. Springer London, 1st edition.

[Ernstsson et al., 2021] Ernstsson, A., Ahlqvist, J., Zouzoula, S., and Kessler, C. (2021). SkePU 3: Portable High-Level Programming of Heterogeneous Systems and HPC Clusters. *International Journal of Parallel Programming*, 49:846–866.

[Fischer et al., 2003] Fischer, J., (auth.), S. G., PhD, F. A. R., and (eds.), S. G. P. (2003). *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag London, 1st edition.

[Francis, 2021] Francis, P. (2021). *Learn Python Programming: A Working Introduction Guide for Python Programming and focuses on the programming language paradigm (process-oriented, object-oriented, function-oriented)*. Personal Edition.

[Frery and Perciano, 2013] Frery, A. C. and Perciano, T. a. (2013). *Introduction to Image Processing Using R: Learning by Examples*. SpringerBriefs in Computer Science. Springer-Verlag London, 1 edition.

[Gamma et al., 1998] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. (1998). *Design Patterns CD. Elements of reusable object-oriented software*. Professional Computing. Addison-Wesley Professional, cdr edition.

[Gan et al., 2022] Gan, L., Wang, Y., Xue, W., and Chau, T., editors (2022). *The 18th International Symposium on Applied Reconfigurable Computing, ARC 2022. Virtual Event, Tsinghua University Beijing, China, September 19-20, 2022*, volume 13569 of *Lecture Notes in Computer Science*. Springer.

[Gebali, 2011] Gebali, F. (2011). *Algorithms and Parallel Computing*. Wiley Series on Parallel and Distributed Computing. Wiley, 1st edition.

[Goossens, 2023] Goossens, B. (2023). *Guide to Computer Processor Architecture: A RISC-V Approach, with High-Level Synthesis*. Undergraduate Topics in Computer Science. Springer, 1 edition.

[Grama et al., 2003] Grama, A., Karypis, G., Kumar, V., and Gupta, A. (2003). *Introduction to Parallel Computing*. Addison Wesley, 2nd edition.

[Hajji et al., 2022] Hajji, B., Mellit, A., and Bouselham, L. (2022). *Practical Guide For Simulation And Fpga Implementation Of Digital Design*. Springer Verlag, Singapor.

[Hauck and DeHon, 2007] Hauck, S. and DeHon, A. (2007). *Reconfigurable computing: the theory and practice of FPGA-based computation*. Systems on Silicon. Morgan Kaufmann, 1st edition.

[Hennessy and Patterson, 2017] Hennessy, J. L. and Patterson, D. A. (2017). *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, sixth edition edition.

[Hwu, 2016] Hwu, W. (2016). *Heterogeneous System Architecture. A New Compute Platform Infrastructure*. Morgan Kaufman, 1st edition.

[Isazadeh et al., 2017] Isazadeh, A., Izadkhah, H., and Elgedawy, I. (2017). *Source Code Modularization: Theory and Techniques*. Springer International Publishing, 1st edition.

[Kaeli et al., 2015] Kaeli, D. R., Mistry, P., Schaa, D., and Zhang, D. P. (2015). *Heterogeneous Computing with OpenCL 2.0*. Morgan Kaufmann, 1st edition.

[Khalgui and Hanisch, 2010] Khalgui, M. and Hanisch, H.-M. (2010). *Reconfigurable Embedded Control Systems: Applications for Flexibility and Agility*. IGI Global, 1 edition.

[Khronos OpenCL Working Group, 2023] Khronos OpenCL Working Group (2023). *The OpenCL Specification, Version 3.0.13*.

[Kirischian, 2016] Kirischian, L. (2016). *Reconfigurable computing systems engineering: virtualization of computing architecture*. CRC Press, 1st edition.

[Koch et al., 2016] Koch, D., Hannig, F., and Ziener, D. (2016). *FPGAs for Software Programmers*. Springer International Publishing, 1st edition.

[Lai et al., 2019] Lai, Y.-H., Chi, Y., Hu, Y., Wang, J., Yu, C. H., Zhou, Y., Cong, J., and Zhang, Z. (2019). HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, pages 242–251, New York, NY, USA. Association for Computing Machinery.

[Lai et al., 2021] Lai, Y.-H., Ustun, E., Xiang, S., Fang, Z., Rong, H., and Zhang, Z. (2021). Programming and Synthesis for Software-defined FPGA Acceleration: Status and Future Prospects. *ACM Transactions on Reconfigurable Technology and Systems*, 14(4):17–39.

[Magoules et al., 2015] Magoules, F., Roux, F.-X., and Houzeaux, G. (2015). *Parallel Scientific Computing*. Iste. Wiley-ISTE, 1 edition.

[Magyari and Chen, 2022] Magyari, A. and Chen, Y. (2022). Review of state-of-the-art fpga applications in iot networks. *Sensors*, 22(19).

[Makino, 2021] Makino, J. (2021). *Principles of High-Performance Processor Design: For High Performance Computing, Deep Neural Networks and Data Science*. Springer, 1st edition.

[McCool et al., 2012] McCool, M., Robison, A. D., and Reinders, J. (2012). *Structured Parallel Programming Patterns for Efficient Computation*. Morgan Kaufmann, 1st edition.

[Murti, 2022] Murti, K. C. S. (2022). *Design principles for embedded systems*. Transactions on computer systems and networks. Springer, 1st edition.

[Nedjah, 2016] Nedjah, N. (2016). *Reconfigurable and adaptive computing : theory and applications*. CRC Press, 1st edition.

[Oh, 2017] Oh, J. (2017). *Operating systems : a multi-perspective episodic approach*. Cognella Academic Publishing, first edition. edition.

[Pacheco and Malensek, 2020] Pacheco, P. and Malensek, M. (2020). *An Introduction to Parallel Programming*. Morgan Kaufmann, 2nd edition.

[Pang and Membrey, 2017] Pang, A. and Membrey, P. (2017). *Beginning FPGA: Programming Metal: Your brain on hardware*. Apress, 1st edition.

[Robey and Zamora, 2021] Robey, R. and Zamora, Y. (2021). *Parallel and High Performance Computing*. Manning Publications, 1st edition.

[Rodríguez et al., 2017] Rodríguez, J., de la Torre, E., and Valdés, M. (2017). *FPGAs: Fundamentals, Advanced Features, and Applications in Industrial Electronics*. CRC Press.

[Schaumont, 2013] Schaumont, P. R. (2013). *A Practical Introduction to Hardware/Software Codesign*. Springer US, 2nd edition.

[Schmidt et al., 2018] Schmidt, B., González-Domínguez, J., Hundt, C., and Schlarb, M. (2018). *Parallel programming: concepts and practice*. Elsevier; MK Morgan Kaufmann Publishers, 1st edition.

[Simpson, 2015] Simpson, P. A. (2015). *FPGA Design: Best Practices for Team-based Reuse*. Springer International Publishing, 2nd edition.

[Snider, 2023] Snider, R. K. (2023). *Advanced Digital System Design using SoC FPGAs: An Integrated Hardware/Software Approach*. Springer, 1st edition.

[Stallings, 2018] Stallings, W. (2018). *Operating Systems: Internals and Design Principles*. Pearson Education Limited, 9th global edition edition.

[Steuwer et al., 2011] Steuwer, M., Kegel, P., and Gorlatch, S. (2011). SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1176–1182.

[Tanenbaum and Austin, 2012] Tanenbaum, A. S. and Austin, T. (2012). *Structured Computer Organization*. Prentice Hall, 6th edition.

[Thakare and Bhandari, 2023] Thakare, A. D. and Bhandari, S. U. (2023). *Artificial Intelligence Applications and Reconfigurable Architectures*. Wiley-Scrivener.

[Tran, 2022] Tran, M. Q. (2022). *The Art of Functional Programming*, volume 86F7F24BF27. Independently published.

[Trussell and Vrhel, 2008] Trussell, H. J. and Vrhel, M., editors (2008). *Fundamentals of Digital Imaging*. Cambridge University Press, 1st edition.

[Waidyasooriya et al., 2018] Waidyasooriya, H. M., Hariyama, M., and Uchiyama, K. (2018). *Design of FPGA-Based Computing Systems with OpenCL*. Springer International Publishing, 1st edition.

[Wilson, 2016] Wilson, P. R. (2016). *Design Recipes for FPGAs, Second Edition: Using Verilog and VHDL*. Newnes is an imprint of Elsevier, 2nd edition.

[Wolf, 2014] Wolf, M. (2014). *High-Performance Embedded Computing. Architectures, Applications, and Methodologies*. Series in Computer Graphics. Morgan Kaufmann, 2nd edition.

[Xu et al., 2022] Xu, S., Huang, W., and Huang, Y., editors (2022). *30th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2022, New York City, NY, USA, May 15-18, 2022*. IEEE Computer Society.

[Zahran, 2019] Zahran, M. (2019). *Heterogeneous Computing: Hardware & Software Perspectives*. Association for Computing Machinery, 1st edition.

[Zhang et al., 2022] Zhang, B., Kannan, R., Prasanna, V., and Busart, C., editors (2022). *Proceeding 32nd International Conference on Field-Programmable Logic and Applications (FPL), Aug. 29 2022 to Sept. 2 2022. Belfast, United Kingdom*. IEEE Computer Society.

ANEXOS

ANEXO A:
Artículo de Investigación con Ponencia y Publicado en Conferencia Internacional en Computación de Alto Rendimiento (HPC)

Proceedings of the Advanced Computing Trends Workshops, 9th Latin America High-Performance Computing Conference, CARLA 2022, Porto Alegre, Brazil, September 26–30, 2022. DOI: https://doi.org/10.5281/zenodo.7469164

Reconfigurable Pipeline Skeleton for Parallel Applications Codesign on High-Performance Heterogeneous Computing Systems

Carlos Alfonso Acosta-León[1][0000−0003−1895−4040]
and Rina Surós[2][0000−0001−6443−9889]

Laboratorio de Computación Heterogénea de Alto Rendimiento (LabCHAR)
Centro de Computación Paralela y Distribuída, Escuela *de* Computación
Universidad Central de Venezuela, Los Chaguaramos, Caracas 1040-A, Venezuela.
cacostal@yahoo.co.uk, carlos.acosta@ucv.ve[1]
rsuros@gmail.com, rina.suros@ciens.ucv.ve[2]

Abstract. Reconfigurable heterogeneous computing systems (RHCS) have been used to exploit parallelism by means of coupled and coordinated processing between FPGA and different microprogrammable computing devices. However, these systems have high programming complexity due to the details associated with parallelism and FPGA logic design. Therefore, the development of the hardware and software components of an application at the same level of abstraction has been difficult to achieve. Several techniques have been described in the literature that attempt to reduce such complexity to the programmer, but without achieving sufficient transparency and abstraction. In this paper we introduce a reconfigurable pattern of parallel 'pipeline' computing, called *PipeSkeleton*. It is an algorithmic skeleton provided as a high-level template in OpenCL code. As a demonstration, a tests suite with different configurations integrating hardware and software components are described. It was shown that configurations with hardware-implemented kernels and software-implemented data input/output run faster and consume fewer resources. As conclusion, the tool provides to the programmer the abstraction level to easily move functionalities between software and hardware during the exploration stage of application design spaces.

Keywords: Heterogeneous Reconfigurable Computing · Parallelism · FPGA · Algorithmic Skeletons · Hardware/Software Co-design.

1 Introduction

This paper focuses on Reconfigurable Heterogeneous Computing Systems (RHCS) based on Field-Programmable Gate Arrays (FPGA) computing devices. Reconfigurable Heterogeneous Computing Systems, described by Zahran in [11], integrate different processing elements, both structurally and functionally, into a single computational system in a way that is transparent to the user. This makes it possible to incorporate specialized processing capabilities to accelerate tasks in

a particular field or area in order to significantly increase performance. However, the problem with RHCS is that they bring high programming complexity [8]. This is because the parallel applications development in these systems requires the integration and coordination of tasks running on software and hardware. This means that the programmer has to pay attention to low-level details associated with parallelism [9], and also to the structural and functional details associated to FPGA logic design [6].

Finally, in this research paper we propose a co-design and programming tool based on reconfigurable algorithmic skeletons. This tool provides the programmer with structured and reconfigurable parallelism for the hardware/software co-design of parallel applications.

2 Related Research Work

In the literature there are works that propose Libraries, APIs or Frameworks of parallel computing patterns encapsulated as algorithmic skeletons. One work oriented to heterogeneous computing systems comes from the SkelCL project [10]. This is a library of algorithmic skeletons implemented using the OpenCL language, portable to different heterogeneous CPU-GPU systems. This library is composed of four parallel skeleton models: Map, Zip, Reduce and Scan, which operate on one-dimensional vectors on CPU and GPU, although not transparently. In the same direction, in a recent work, Ernstsson et al. propose SkePU [4], an open source CPU-GPU heterogeneous system programming framework for multicore CPUs and multi-GPU systems. They are C++ templates with data parallelism skeletons, with support for execution on multi-GPU systems with both CUDA and OpenCL and clustered execution. However, these projects do not incorporate hardware skeletons for heterogeneous CPU-GPU-FPGA architectures.

The HeteroCL project by Lai et al. [7] is one among few works where a programming infrastructure composed of a domain-specific language (DSL) based on Python with CPU, GPU and FPGA oriented compilation is proposed. It provides abstraction through programming that decouples the algorithm specification from the computing architecture hardware. Also, it allows the programmer to explore performance systematically with hardware implementations using systolic array templates and dataflow stencils. However, the tool does not consider the hardware/software tasks co-design of a parallel application at the same level of abstraction.

Based on the reviewed works, there are few oriented to facilitate hardware/software co-design and programming of system-level parallel applications on heterogeneous computing platforms with CPU-GPU-FPGA. In addition, some reviewed works assume that the programmer has some knowledge of reconfigurable computing system architecture, details associated with parallelism and FPGA design.

3 Preliminary Background

Although there are several high-level abstraction techniques for design and programming, in this research work we are interested in those that allow hiding and encapsulating parallelism. From these design and programming techniques, Cole's algorithmic skeletons and the hardware/software co-design have been chosen. They are the foundation for the design of the proposed tool that integrates them into a solution or framework called SkeletonCoRe.

Applications development for heterogeneous architecture computing systems involve code sections that are usually designed and implemented separately in hardware and software, using different tools and techniques, and by different people. So, Co-design [1, 5] is a methodology that integrates the cooperative design of hardware and software sections of an application. However, it does not guarantee that these sections are developed at the same level of abstraction, which makes it difficult to migrate functionality between software and hardware. As a solution to this problem, we have proposed the use of algorithmic skeletons in the co-design process. This approach allows the development of such components at the same level of abstraction and the transparent movement of functionalities between CPUs, GPUs and FPGAs.

Cole [2, 3] introduced Algorithmic Skeletons as a mechanism for encapsulating and reusing parallelism. They are a abstraction structured approach that allows separating a parallel computing pattern from its implementation in a particular computing architecture. Skeletons hide the explicit details of parallel programming from the programmer and make possible an appropriate and efficient solution to a specific problem.

4 PipeSkeleton: A Reconfigurable Pipeline Skeleton

Algorithmic skeletons, hardware/software co-design, FPGAs and the OpenCL language are integrated into a framework, called SkeletonCoRe. This high-level framework provides a catalog of reconfigurable algorithmic skeletons oriented to the parallel applications co-design and programming on reconfigurable heterogeneous computing systems. Here, a skeleton is implemented as a higher-order function through sequential code templates, with data and functions as parameters, coded using OpenCL language. These parameters internally define its parallel behavior. Each skeleton transparently coordinates the interaction between software tasks on CPU-GPU and hardware tasks on FPGA.

As a proof-of-concept, we introduce and evaluate the cost of a reconfigurable algorithmic skeleton, denoted as "*PipeSkeleton* "Reconfigurable Pipeline Skeleton". This algorithmic skeleton is an instance from the reconfigurable skeletons catalog of the SkeletonCoRe Framework.

This skeleton implements the well-known pipelined parallel processing pattern called *"pipeline"* in which a "stream" of data elements flows through a sequence of stages (ordered like a pipeline) where in each one an operation or task is applied to each element of the stream. This behavior can be described

as follows, $f : \alpha \rightarrow \beta$ on a "stream" of input values $a_1, a_2, \ldots, a_n$ which implies a composition over n functions $f_1 \ldots f_n$ such that $f_1 : a_1 \rightarrow \gamma_1, \ldots, f_i : \gamma_{i-1} \rightarrow \gamma_i, \ldots, f_n : \gamma_{n-1} \rightarrow \beta$. The idea is to exploit parallelism from processing on different elements of the "input stream" (without data dependencies).

PipeSkeleton is a higher-order template that is instantiated with several parameters, between data and behavior (As is shown for the pseudocodes in Fig. 1). That is, the size of the hardware and software partitions, an array of tasks for each pipeline stage and the data segments size. This skeleton manages the pipeline through two partitions, one in software (CPU-GPU) and one in hardware (FPGA), both with defined sizes and whose sum conforms the total length of the pipeline. To each pipeline stage corresponds a specific task for processing, including input and output tasks. The communication interface that connects the software partition on the host computer with the hardware partition on the FPGA board is transparent to the programmer.

```
include  ``opencl.h''
include  ``skeletoncore.h"
/* Global declaration */
FILE *List_imagesIn, *List_imagesOut;
int segment_size;
main (argc, argv[ ]) {
 /* Pipeline size */
 int cpu_pipesize, gpu_pipesize, fpga_pipesize;

 /* Create heterogeneous pipeline */
 device_type *device_cpu,*device_gpu,*device_fpga;
 func_type *tasks_List[cpu_pipesize+gpu_pipesize+fpga_pipesize];
 /*------- Program body -------*/
 /* Setting up the SkeletonCoRe Framework */
 SkeletonsCoRe_init}();
 /* Getting the PipeSkeleton size */
 skc_read(cpu_pipesize, gpu_pipesize, fpga_pipesize);
 if (cpu_pipesize > 0) | (gpu_pipesize > 0) | (fpga_pipesize > 0)  {
       /* Getting the tasks for the stages of PipeSkeleton */
       skc_getTasks(tasks_list[task_1, task_2, ..., task_n]);
       /* Reading the image source file */
       skc_getData(List_imagesIn);
       /* Creates instance of PipeSkeleton */
       myPipe = new skc_PipeSkeleton}();
       /* Invoke PipeSkeleton for image processing */
       myPipe(*device_cpu, *device_gpu, *device_fpga, *tasks_List[],
              *List_imagesIn, *List_imagesOut, segment_size);
       /* Writing the output image file */
       skc_putOutcomes(List_imageOut);
 }
 else {
       printf("Pipeline not defined!");
      }
 /* Finalizing the SkeletonsCoRe Framework */
 SkeletonCoRe_finalize}();
}
```

a) Source pseudocode for using PipeSkeleton

```
void PipeSkeleton(*device_cpu, *device_gpu, *device_fpga, *tasks_List[],
                  *List_imagesIn, *List_imagesOut, segment_size) {
cl_uint num_devices_returned;
cl_device_id devices[3];
channel float cpu_gpu_channel, gpu_fpga_channel, fpga_cpu_channel;
func_type *cpu_Tasks, *gpu_Tasks, *fpga_Tasks;
cpu_Task = &tasks_List[cpu_pipesize];
gpu_Task = &tasks_List[gpu_pipesize];
fpga_Task = &tasks_List[fpga_pipesize];

/* Get the device(s) */
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &devices[0], &num_devices_returned);
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &devices[1], &num_devices_returned);
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_FPGA, 1, &devices[2], &num_devices_returned);

/* Create contexts for devices */
cl_context context;
context = clCreateContext(0, 3, devices, NULL, NULL, NULL, &err);

/* Create commands queue for devices */
cl_command_queue queue_gpu, queue_cpu, queue_fpga;
device_gpu = clCreateCommandQueue(context, devices[0], 0, &err);
device_cpu = clCreateCommandQueue(context, devices[1], 0, &err);
device_fpga = clCreateCommandQueue(context, devices[2], 0, &err);

/* Basic pipeline for PieSkeleton */
__kernel void cpu(__global int* in) {
  for (int i = 0; i < workload_size; ++i) {
     int *image_segment = read_Image();        // Get images by segments
     value[segment_size] = cpu_Tasks(&image_segment, 0, 255); // do some work
     write_channel(cpu_gpu_channel, &value);    // send data to the next partition
  }
}

__kernel void gpu() {
  for (int i = 0; i < workload_size; ++i) {
     int value = read_channel(cpu_gpu_channel);  // take data from cpu
     value[segment_size] = gpu_Tasks(&image_segment, 0, 255);  // do some work
     write_channel(gpu_fpga_channel, &value);    // send data to the next partition
  }
}

__kernel void fpga(__global int* out) {
  for (int i = 0; i < workload_size; ++i) {n
     int value = read_channel(gpu_fpga_channel);         // take data from gpu
     value[segment_size] = fpga_Tasks(&image_segment, 0, 255); // do some work
     write_channel(fpga_cpu_channel, &value);            // write result in the end
  }
}

/* Start concurrently tasks (kernels) for partitions in CPU-GPU-FPGA */
clEnqueueTask(device_cpu, *TasksList[cpu_size]);
clEnqueueTask(device_gpu, *TasksList[gpu_size]);
clEnqueueTask(device_fpga,*TasksList[fpga_size]);

clFinish(device_fpga); // last kernels in our pipeline

}
```

b) OpenCL pseudocode for implementing PipeSkeleton

Fig. 1. Examples of OpenCL pseudocode for PipeSkeleton.

5 Functionality tests for PipeSkeleton

5.1 Development Platform specifications

To evaluate our tool we have used a reconfigurable heterogeneous computing platform consisting of an AsRock motherboard supporting an Intel® Core™ 10th Gen i5-1040F processor with 6 cores (12 threads) at 2.9 GHz and two banks of DDR4 memory of 8 GB each at 2933 MHz. The motherboard is equipped with an Intel® Stratix® 10 GX FPGA board with 10.2 million Logic Elements (LEs) and an NVIDIA GeForce GTX 1060 graphics card with 1280 cores and 6GB of memory. All these devices are connected via a 64-bit system bus. In addition,

the heterogeneous computing system uses Linux Ubuntu version 20.04.4 LTS, and GCC version 9.4.0 to compile the Host (CPU-GPU) and FPGA codes in openCL language version 1.0. The application development environment for the heterogeneous system is Quartus version 22.1.0 with Intel FPGA SDK for OpenCL version 22.1. Based on the architecture of the SkeletonCoRe framework we have chosen Intel® Quartus® Prime Design Software and Intel FPGA SDK for OpenCL because they cover the four lower layers and therefore provide the functionalities associated with the process of compiling the components for the CPU and GPU (scheduling) and the synthesis (partitioning, placement and routing) of the components for the FPGA.

5.2 Description of Functionality Tests

A set of experiments have been prepared to test the functionality and performance of PipeSkeleton. We have implemented a parallel application composed of a queue of image operators whose kernels are assigned as tasks to each stage of the pipeline. The order of precedence of these image operators is fixed, and they are as follows: → Interpolation → Sobel's Edge Detection → Sum → Thresholding → Sum →. In addition, the test workload is composed of four lists of 256 images each with dimensions of 256x256, 512x512, 1024x1024 and 2048x2048 pixels, respectively, to exploit data parallelism.

In Fig. 2 can be seen the configurations used to perform the performance and functionality test of *PipeSkeleton*. As test configurations (experiments), pipeline partitions (stages) of the form CPU-GPU-FPGA into our heterogeneous computing system have been performed. So, as an initial and reference configuration for performance comparison, the entire pipeline of our parallel application (five stages) has been mapped as software tasks only on CPU (5-0-0 or CPU-0-0), instead of the best sequential code. Then, only on GPU (0-5-0 or 0-GPU-0) (see Fig. 2a) and also, the entire pipeline has been mapped as hardware tasks only on the FPGA device (0-0-5 or 0-0-FPGA) (see Fig. 2b).

Finally, a intermediate configuration combinations of partitions with software and hardware tasks or kernels of the pipeline have been distributed and mapped between the GPU and the FPGA (0-GPU-FPGA) respectively, always keeping the precedence order of the image operators in the pipeline (see Fig. 2c). In all test configurations, both benchmark and intermediate, the data read and write tasks have been fixedly mapped to CPU only.

5.3 Funtionality Test Results

The results of the tests are shown in Table 1 and introduces the processing times of four lists of 256 images of dimensions 256x256, 512x512, 1024x1024 and 2048x2048 pixels. Each image is divided into two-line or two-row image data segments for processing. Processing times are observed when PipeSkeleton is implemented on CPU only (5-0-0). These times are taken as a reference in determining the computational speedup with respect to the times of PipeSkeleton configured on GPU only (0-5-0), FPGA only (0-0-5) and hardware/software partitioning combinations between GPU and FPGA (0-X-X).

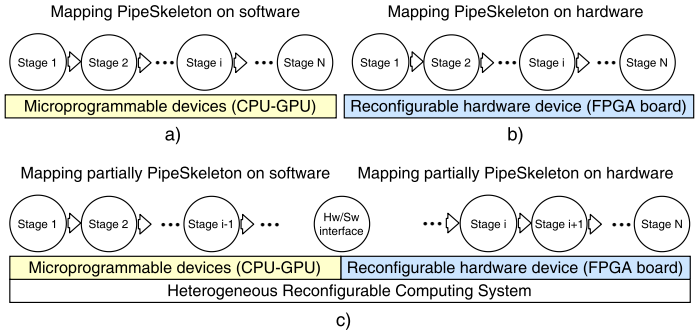


Fig. 2. Mapping PipeSkeleton on the heterogeneous reconfigurable computing system.

Table 1. Comparison of processing time for computing devices CPU, GPU and FPGA.

Partition (CPU-GPU-FPGA)	Image size (pixels)	Segments x image	Workload (segments x all images)	Total time x segment (milisec)	Total time x image (milisec)	- With skeleton - Total time x workload (miliseconds)	- No skeleton - Total time x workload (miliseconds)	Skeleton overhead (milisec)	Total Cost of FPGA LUTs	Speed-Up with skeleton %
5-0-0	256 x 256	128	32768	1.640625	43.312500	10753.312500	10753.204967	0.107533	0	**1.00**
	512 x 512	256	65536	2.985938	155.268750	39139.668750	39139.277353	0.391397	0	**1.00**
	1024 x 1024	512	131072	5.434406	560.830725	142464.046725	142462.622085	1.424640	0	**1.00**
	2048 x 2048	1024	262144	9.890619	2033.511344	518561.217584	518556.031971	5.185612	0	**1.00**
0-5-0	256 x 256	128	32768	0.032471	0.857227	212.825977	212.823848	0.002128	0	**50.53**
	512 x 512	256	65536	0.058447	3.039258	766.126758	766.119097	0.007661	0	**51.09**
	1024 x 1024	512	131072	0.105205	10.857164	2757.972164	2757.944584	0.027580	0	**51.66**
	2048 x 2048	1024	262144	0.199890	41.097312	10480.134312	10480.029510	0.104801	0	**49.48**
0-4-1	256 x 256	128	32768	0.025618	0.828670	207.219295	207.217223	0.002072	1023	**51.89**
	512 x 512	256	65536	0.045607	2.915727	737.666352	737.658975	0.007377	2044	**53.06**
	1024 x 1024	512	131072	0.081193	10.318853	2626.031078	2626.004818	0.026260	4084	**54.25**
	2048 x 2048	1024	262144	0.155573	39.506656	10083.841600	10083.740761	0.100838	8159	**51.42**
0-3-2	256 x 256	128	32768	0.022633	0.636413	158.382601	158.381017	0.001584	7161	**67.89**
	512 x 512	256	65536	0.040450	2.234108	563.810536	563.804898	0.005638	14307	**69.42**
	1024 x 1024	512	131072	0.072294	7.897029	2007.109111	2007.089040	0.020071	28586	**70.98**
	2048 x 2048	1024	262144	0.125386	28.321994	7225.485488	7225.413234	0.072255	57115	**71.77**
0-2-3	256 x 256	128	32768	0.021395	0.605161	150.637599	150.636092	0.001506	8184	**71.39**
	512 x 512	256	65536	0.038328	2.124716	536.240194	536.234831	0.005362	16351	**72.99**
	1024 x 1024	512	131072	0.068663	7.510770	1908.961870	1908.942780	0.019090	32670	**74.63**
	2048 x 2048	1024	262144	0.115311	26.337161	6719.445032	6719.377838	0.067194	65274	**77.17**
0-1-4	256 x 256	128	32768	0.020748	0.589507	146.765070	146.763602	0.001468	14322	**73.27**
	512 x 512	256	65536	0.037257	2.070011	522.455013	522.449789	0.005225	28615	**74.91**
	1024 x 1024	512	131072	0.066903	7.317696	1859.888305	1859.869706	0.018599	57172	**76.60**
	2048 x 2048	1024	262144	0.108099	23.333340	5951.559289	5951.499773	0.059516	114229	**87.13**
0-0-5	256 x 256	128	32768	0.020337	0.536895	133.296270	133.294937	0.001333	18414	**80.67**
	512 x 512	256	65536	0.036606	1.903535	479.837285	479.832487	0.004798	36790	**81.57**
	1024 x 1024	512	131072	0.065892	6.800013	1727.361513	1727.344240	0.017274	73507	**82.47**
	2048 x 2048	1024	262144	0.106744	21.946648	5596.565908	5596.509942	0.055966	146866	**92.66**

Legend: For our workload we get the experimental data from four list of 256 images each x 2 lines x image segment.

6 Results Discussion

Based on the results obtained and shown in Table 1, a significant increase of Speed-Up is observed in the GPU benchmark tests up to 50%, and FPGA benchmark tests up to 92%, with respect to the execution times of PipeSkeleton running only on CPU. Likewise, it is observed that intermediate tests with combinations of PipeSkeleton partitions with fewer stages on GPU and more stages on FPGA show a progressive and significant increase of Speed-Up between 51.42% and 87.13% with respect to CPU times. These tests also demonstrate the possibility to explore parallel application design spaces and choose the configuration with the best performance and efficiency.

However, we also noticed a small difference (overhead) between the processing times when using the encapsulation provided by the PipeSkeleton algorithmic

skeleton (which provides implicit parallelism) with respect to when using the parallel application without the algorithmic skeleton (explicit parallelism).

Similarly, there is an additional hardware resource cost in logic elements (4-input LUTs, without optimization) when implementing the PipeSkeleton partition on FPGA. The implementation of filters, registers and data vectors of the size of the image segment to be processed is the reason for the high utilization of logic elements (4-input LUTs) of the FPGA. Moreover, the cost of FPGA LUTs is proportional to the size of the problem, i.e., to the size of the image segments or vectors and to the complexity of the tasks or kernels of the pipeline partitions that are implemented in the FPGA.

Another observation, obtained from Table 1, refers to the fact that the pipeline stage with the highest latency constituted an appreciable bottleneck in processing on CPU and GPU, but very little on FPGA. Also, we consider that the 1280-core GPU limit (NVIDIA GeForce GTX 1060 card) reduced the processing time of the 2048x2048 pixel image list, where the Speed-Up was reduced from 50.53% to 49.48%.

Finally, since the CPU and both the GPU and FPGA boards are connected to the heterogeneous computing system via a 64-bit high-speed bus, low communication latency was observed in the data transfer of the 256 images used in each test image list.

7 Conclusions and Future Work

We have shown the use and utility of PipeSkeleton. Skeleton approach has allowed to implement parallelism and reconfiguration easily, transparently and independently of the available computing platform with high performance.

Tests have been performed with different configurations integrating hardware and software tasks. In each configuration we have measured the cost in execution time and amount of resources used. It has been shown that configurations with FPGA-implemented kernels and CPU-implemented data input/output tasks run faster and consume fewer resources.

In addition, the time overhead and FPGA resources used in the implementation of the skeleton is relatively small. Although the resources allocated by the language to encapsulate the parallel pattern using a template with parameters makes this cost almost constant and independent of the size of the problem managed by the PipeSkeleton skeleton.

However, the cost of the abstraction provided by the skeletons may result in a small loss of efficiency compared to explicitly parallel code written by experts. Still, the skeleton abstraction can lead to higher productivity and performance because the parallelism structuring can enable automated optimizations. In addition, this abstraction provides the programmer with the ability to easily move functionality between software and hardware during the exploration stage of application design spaces.

As additional work, it is necessary to extend the library of skeletons that implement other parallel computing patterns such as map & reduce, divide &

conquer, tasks farm, data farm, etc. Also, tests should be performed with more demanding applications in terms of computing power, such as those in the area of cryptography or numerical simulation, for example, to evaluate the functionality and stress the performance of the skeletons.

Acknowledgements The first author of this article is very grateful to his supervisor, Dr. Murray I. Cole, for his invaluable mentoring, help and support during his PhD in Informatics at the University of Edinburgh, Scotland-UK. He would also like to thank his co-supervisor Dr. Rina Surós at the UCV for her unconditional support.

References

1. Cardoso, J.M.P., (auth.), M.H.: Reconfigurable Computing: From FPGAs to Hardware/Software Codesign. Springer-Verlag New York, 1st edition edn. (2011)
2. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press & Pitman, 1 edn. (1989)
3. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Computing **30** (2004)
4. Ernstsson, A., Ahlqvist, J., Zouzoula, S., Kessler, C.: SkePU 3: Portable High-Level Programming of Heterogeneous Systems and HPC Clusters. International Journal of Parallel Programming **49**, 846–866 (2021). https://doi.org/10.1007/s10766-021-00704-3
5. Ha, S., Teich, J.: Handbook of Hardware/Software Codesign. Springer Netherlands, 1st edn. (2017)
6. Hajji, B., Mellit, A., Bouselham, L.: Practical Guide For Simulation And Fpga Implementation of Digital Design. Springer Verlag, Singapor (2022)
7. Lai, Y.H., Chi, Y., Hu, Y., Wang, J., Yu, C.H., Zhou, Y., Cong, J., Zhang, Z.: HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In: Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. p. 242–251. FPGA '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3289602.3293910
8. Lai, Y.H., Ustun, E., Xiang, S., Fang, Z., Rong, H., Zhang, Z.: Programming and Synthesis for Software-defined FPGA Acceleration: Status and Future Prospects. ACM Transactions on Reconfigurable Technology and Systems **14**(4), 17–39 (2021). https://doi.org/10.1145/3469660
9. Pacheco, P., Malensek, M.: An Introduction to Parallel Programming. Morgan Kaufmann, 2nd edn. (2020)
10. Steuwer, M., Kegel, P., Gorlatch, S.: SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum. pp. 1176–1182 (2011). https://doi.org/10.1109/IPDPS.2011.269
11. Zahran, M.: Heterogeneous Computing: Hardware & Software Perspectives. Association for Computing Machinery, 1st edn. (2019)

ANEXO B: Herramientas de Hardware y Software utilizadas en el Desarrollo de la Tesis

B.1 Características del Microprocesador Intel® Core™ 10ma Gen i5-10400F

El Intel Core i5-10400F es un procesador de gama media para computadores de escritorio de seis núcleos basado en la arquitectura Comet Lake (CML-S, 4ª generación de Skylake) con un socket de procesador LGA 1200 (Socket H5). El procesador funciona a una frecuencia de entre 4,1 y 4,8 GHz y puede ejecutar hasta 12 hilos simultáneamente gracias a la tecnología Hyper-Threading de Intel. El procesador se sigue fabricando en el antiguo proceso de 14nm (14nm++). La arquitectura del Comet Lake es similar a la del Coffee Lake y ofrece las mismas características y se produce en el mismo proceso de 14nm.

Además de las mejoras en la velocidad del reloj, el controlador de memoria ahora también soporta una SDRAM DDR4-2933 más rápida, con doble canal y capacidad de memoria interna máxima de 128 GB. Gracias a las altas velocidades de reloj (y a los posibles ajustes de TDP con la mayoría de las placas base), el Core i5-10400F ofrece un buen rendimiento en juegos y gracias a los 6 núcleos también un buen rendimiento en aplicaciones. Soporta un set de instrucciones: SSE4.1, SSE4.2, AVX 2.0. Además, Intel especifica el TDP a 65 vatios (PL1), pero bajo carga completa se consumen hasta 134 vatios (PL2) por hasta 28 segundos (Tau). Las caracteristicas mas relevantes de su tecnología son Ver Fig. B.1:

- **Compatible con la memoria Intel® Optane™:** La memoria Intel® Optane™ es un nuevo y revolucionario tipo de memoria no volátil que se encuentra entre la memoria del sistema y el almacenamiento con el fin de acelerar el desempeño y la capacidad de respuesta del sistema. Al combinarse con el controlador de la Tecnología de almacenamiento Intel® Rapid, administra de manera fluida varios niveles de almacenamiento al mismo tiempo que presenta una sola unidad virtual al sistema operativo, lo cual permite que los datos de uso frecuente residan en el nivel de almacenamiento más rápido. La memoria Intel® Optane™ requiere de configuración específica del hardware y el software.
- **Versión de la tecnología Intel® Turbo Boost:** La Tecnología Intel® Turbo Boost aumenta dinámicamente la frecuencia del procesador cuando sea necesario sacando provecho de la ampliación térmica y de energía para que tenga un impulso en la velocidad cuando lo necesite, y un aumento en la eficacia energética cuando no.
- **Tecnología Hyper-Threading Intel®:** La Tecnología Intel® Hyper-Threading ofrece dos cadenas de procesamiento por núcleo físico. Las aplicaciones con muchos subprocesos pueden realizar más trabajo en paralelo, completando antes las tareas.
- **Tecnología de virtualización Intel® (VT-x):** La tecnología de virtualización (VT-x) Intel® permite que una plataforma de hardware funcione como varias plataformas "virtuales". Ofrece mejor capacidad de administración limitando el tiempo de inactividad y manteniendo la productividad a través del aislamiento de las actividades de cómputo en particiones separadas.

- **Tecnología de virtualización Intel® para E/S dirigida (VT-d):** La Tecnología de virtualización Intel® para E/S dirigida (VT-d) continúa desde la compatibilidad existente para virtualización de IA-32 (VT-x) y el procesador Itanium® (VT-i), sumando nuevas compatibilidades para virtualización de dispositivos de E/S. Intel VT-d puede ayudar a los usuarios finales a mejorar la seguridad y la confiabilidad de los sistemas y también a mejorar el desempeño de los dispositivos de E/S en un entorno virtualizado.
- **Intel® VT-x con tablas de páginas extendidas (EPT):** Intel® VT-x con Tablas de página extendidas (EPT), también conocidas como Traducción de direcciones de segundo nivel (SLAT), brinda aceleración a las aplicaciones virtualizadas con uso intensivo de memoria. Las Tablas de página extendidas en las plataformas de Tecnología de virtualización de Intel® reducen los costos adicionales de memoria y alimentación, y aumentan el rendimiento de la batería mediante la optimización del hardware de la administración de la tabla de página.
- **Intel® 64:** La arquitectura Intel® 64 ofrece procesamiento informático de 64 bits en plataformas para servidores, estaciones de trabajo, PC y portátiles cuando se la combina con software compatible.[1] La arquitectura Intel 64 mejora el desempeño permitiendo que los sistemas direccionen más de 4 GB de memoria física y virtual.
- **Conjunto de instrucciones:** Una serie de instrucciones hacen referencia al conjunto básico de comandos e instrucciones que un microprocesador comprende y puede llevar a cabo. El valor que se muestra representa con qué conjunto de instrucciones de Intel es compatible este procesador.
- **Extensiones de conjunto de instrucciones:** Las extensiones de conjunto de instrucciones son instrucciones adicionales que pueden aumentar el rendimiento cuando se realizan las mismas operaciones en múltiples objetos de datos. Estas pueden incluir a SSE (Streaming SIMD Extensions) y AVX (Advanced Vector Extensions).
- **Estados de inactividad:** Los estados de inactividad (estados C) se utilizan para ahorrar energía cuando el procesador esté inactivo. C0 es el estado operacional, lo que significa que la CPU está funcionando correctamente. C1 es el primer estado de inactividad, C2 el segundo, etc., donde se realizan más acciones de ahorro de energía para estados C con valores numéricos más altos.
- **Tecnología Intel SpeedStep® mejorada:** La tecnología Intel SpeedStep® mejorada es un medio avanzado para permitir un desempeño muy alto y a la vez satisfacer la necesidad de conservación de energía de los sistemas portátiles. La tecnología Intel SpeedStep® tradicional conmuta el voltaje y la frecuencia en tándem entre niveles altos y bajos en respuesta a la carga del procesador. La Tecnología Intel SpeedStep® mejorada se desarrolla en esa arquitectura utilizando las estrategias de diseño como separación entre cambios de voltaje y frecuencia, y partición de reloj y recuperación.
- **Tecnologías de monitoreo térmico:** Las tecnologías de monitor térmico protegen el paquete y el sistema del procesador de fallas térmicas a través de varias funciones de administración térmica. Un Sensor digital térmico (DTS) en matriz detecta la temperatura del núcleo, y las funciones de administración térmica reducen el consumo de energía del paquete y, por lo tanto, la temperatura cuando se requiere para mantener normales los límites de operación.
- **Tecnología Intel® Identity Protection:** La tecnología de protección de la identidad Intel® es una tecnología de token de seguridad integrada que ayuda a proporcionar un método simple, resistente a las

alteraciones para proteger el acceso a su cliente y datos de negocio de amenazas y fraudes. La tecnología de protección de la identidad Intel® proporciona pruebas basadas en el hardware de una PC de usuario único a sitios web, instituciones financieras y servicios de red, lo que verifica que intentar ingresar no es malware. La tecnología de protección de la identidad Intel® puede ser un componente clave en las soluciones de autenticación de dos factores para proteger su información en sitios web y cuentas de negocios.

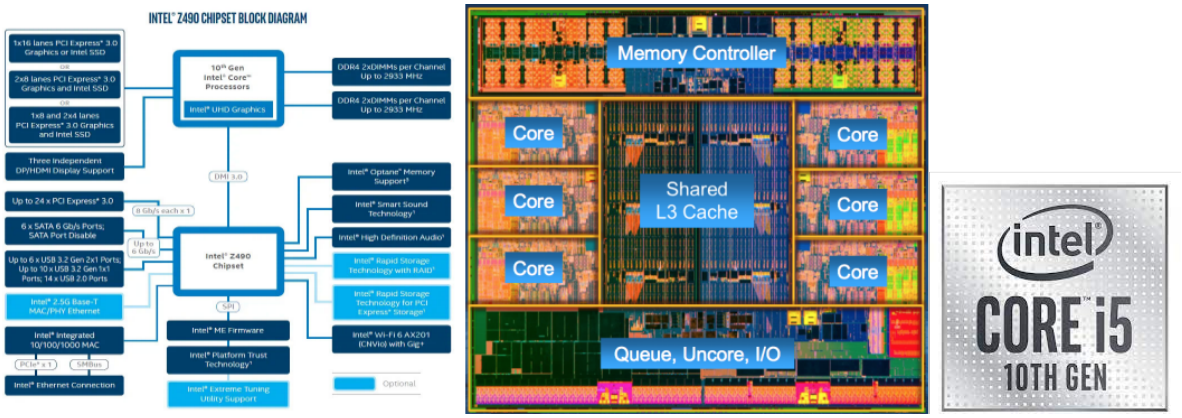


Figura B.1: Chipset y Modelo general de la arquitectura de un microprocesador Intel Core i5. Fuente: www.intel.com

B.2 Características de la Tarjeta Gráfica NVIDIA® GeForce GTX 1060

Las especificaciones técnicas de la GTX 1060 comprende una GPU GP106 con 1280 CUDA Cores funcionando a una frecuencia de Boost de 1.700 MHz, con 6 GB de memoria GDDR5 y una velocidad de 8.000 MHz y cuenta con salidas HDMI 2.0B, un DVI de doble enlace y tres DisplayPort. Las características mas relevante de ésta tecnología de GPU son (Ver Fig. B.2):

- **Arquitectura Pascal:** La terjeta GeForce GTX 1060, está basada en la arquitectura Pascal y un proceso de fabricación es de 16 nm FinFET (TSMC) con consumo de 120 vatios, la cual incorpora 1280 sombreadores (shaders), 80 unidades de textura (TMUs), 48 unidades de renderizado (ROPs). En cuanto a memoria VRAM ésta versión de 6 GB es del tipo GDDR5 de 8 GHz a 192 bits.
- **Rendimiento:** Respecto a su rendimiento tiene una tasa de texturas entre 130 y 150 Gtexel/s, una tasa de píxeles entre 80 y 90 Gpixel/s, una potencia de cómputo de 4 a 5 Tflops aproximados y un ancho de banda de memoria desde 192 GB/s y más allá de los 210 GB/s aproximados.
- **Programación de Carga Dinámica:** Esto permite que el programador ajuste dinámicamente la cantidad de GPU cores asignada a varias tareas, lo que garantiza que la GPU permanezca saturada de trabajo, excepto cuando no haya más trabajo que pueda distribuirse de forma segura. Por lo tanto, Nvidia ha habilitado de forma segura la computación asíncrona en el controlador de Pascal.
- **Paralelismo:** Además de todo lo anterior es posible, con su tecnología CUDA y su compatibilidad con OpenCL, realizar muchas tareas en paralelo de grano fino (paralelismo de datos principalmente), liberando a la CPU de trabajo, haciéndola ideal para edición de imágenes, fotografía, videojuegos, física de objetos, realidad virtual y una gran cantidad de análisis de una gran cantidad de datos como puede ser el estudio climático, biomedicina o investigación, desarrollo y fabricación de nuevos materiales, etc.

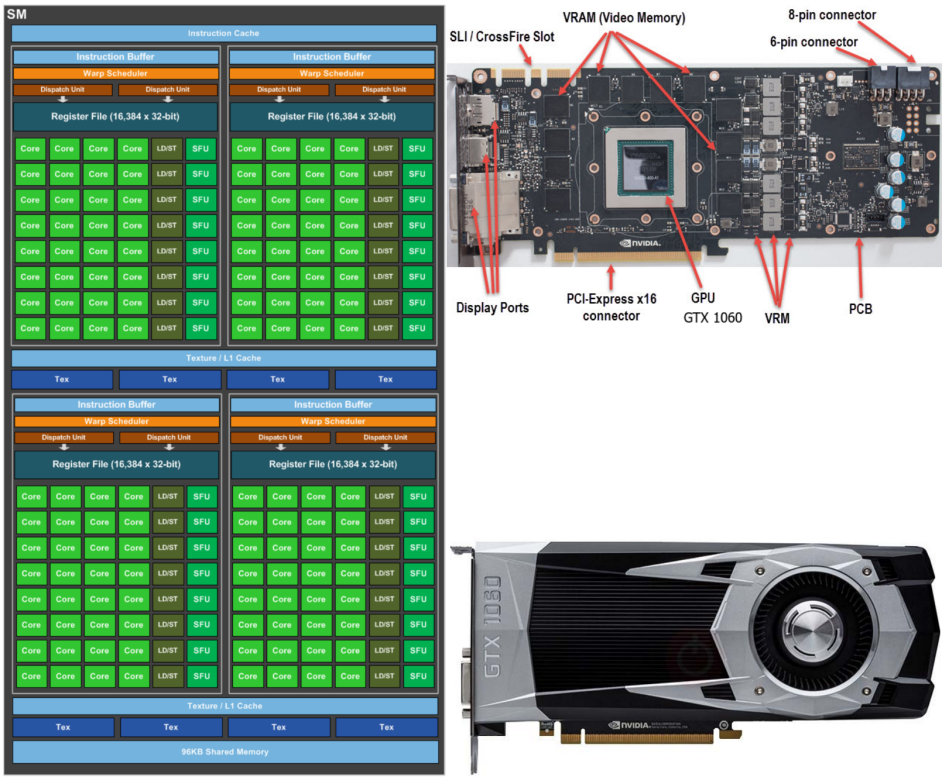


Figura B.2: Modelo general de la Arquitectura Pascal de Nvidia 1060 GTX.
Fuente: www.nvidia.la.com

B.3 Características de la Tarjeta FPGA Intel® Stratix® 10GX

La FPGA Intel® Stratix® 10GX es un FPGA de sistema integrado en chip que innova en el desempeño, la eficiencia en el consumo de energía, la densidad y la integración del sistema. Los dispositivos Intel® Stratix® 10GX cuentan con la arquitectura de FPGA Intel® Hyperflex™ y están construidos con la combinación de la tecnología, patentada por Intel, de puente de interconexión de chips múltiples integrado (EMIB), el bus de interfaz avanzada (AIB), chipsets; y ofrecen hasta el doble de ganancia de desempeño en comparación con las FPGA de alto desempeño de la generación anterior de Intel.

Esta tecnología de FPGA Stratix 10GX, está compuesta por un total de 43.300 millones de transistores que componen sus millones de elementos lógicos (LEs), cifra que le da parte del nombre a este chip.

También cuentan con hasta 96 transceptores de propósito general en mosaicos con transceptor separados, y desempeño de interfaz de memoria externa DDR4 de 2666 Mbps. Los transceptores tienen capacidad para hasta 28,3 Gbps en corto alcance y en el plano posterior. Estos dispositivos han sido optimizados para aplicaciones de FPGA que requieren el ancho de banda de transceptor más alto y un desempeño de trama central mayor. Las características mas relevante de ésta tecnología FPGA son Ver Fig. B.3:

- **Arquitectura FPGA Intel® Hyperflex™:** Introduce registros adicionales que rodean todas partes en la trama de FPGA. Estos registros adicionales, llamados hiperregistros, están disponibles en cada segmento de enrutamiento de interconexión y en las entradas de todos los bloques funcionales de computación. Los hiperregistros posibilitan tres técnicas de diseño clave para lograr la duplicación

del desempeño de núcleo: La hiperresincronización (Hyper-Retiming) para eliminar vías críticas, la hiperconducción (Hyper-Pipelining) para eliminar los retrasos de enrutamiento y la hiperoptimización (Hyper-Optimization) para lograr mejor desempeño.

Por otro lado, el chipsets utiliza la tecnología de sistema en paquete 3D de Intel. Esta tecnología es el puente de interconexión de chips múltiples integrada que ofrece un flujo de integración simple y ofrece una interconexión de ultra alta densidad entre chips heterogéneos del mismo paquete. También permite funciones en el paquete que eran demasiado complejas o tenían costos prohibitivos para implementar con soluciones de integración.

- **Tecnología de empaquetado Intel EMIB para dispositivos Intel® Stratix® 10 (Embedded Multi-Die Interconnect Bridge, EMIB):** Patentada de Intel permite la integración efectiva en el paquete de componentes críticos del sistema, como analógico, memoria, ASIC, CPU, etc. La tecnología EMIB ofrece un flujo de fabricación más simple en comparación con otras tecnologías de integración en paquetes. Además, EMIB elimina la necesidad de usar a través de vías de silicio (TSV) y silicio de interposición especializado, lo que permite una solución que ofrece un mayor desempeño, menos complejidad y una integridad superior de la señal y la energía.

  La tecnología EMIB utiliza un pequeño chip de silicio incrustado en el sustrato para proporcionar una interconexión de ultra alta densidad entre los dados (die). El chip de control de voltaje estándar conecta la energía y las señales del usuario desde el chip a las bases del paquete. Este enfoque minimiza la interferencia debido al ruido de conmutación del núcleo y la diafonía para brindar una señal superior e integridad de energía.

- **Los Transceptores de Intel® Stratix® 10 FPGA:** Son dispositivos de sistema integrados en chip FPGA que ofrecen una nueva era de tecnología con la introducción de transceptores innovadores heterogéneos de sistema en paquete (SiP) 3D. Los mosaicos del transceptor se combinan con una estructura central programable y monolítica que utiliza la integración del sistema en el paquete para abordar las demandas cada vez mayores de ancho de banda del sistema en prácticamente todos los segmentos de aplicación. Los mosaicos de transceptor permiten el mayor número de canales de transceptor FPGA sin sacrificar la facilidad de uso.

- **Interfaces de memoria paralela:** Los dispositivos Intel® Stratix® 10 ofrecen compatibilidad con memoria paralela de hasta 2666 Mbps para SDRAM DDR4 y admiten una amplia gama de otros protocolos que se muestran a continuación. El controlador de memoria dura ofrece un alto desempeño con bajo consumo de energía, incluida la compatibilidad con: DDR4, DDR3/DDR3L, LPDDR3.

  Además, el soporte de controlador soft ofrece flexibilidad para admitir una amplia gama de estándares de interfaz de memoria, incluidos: RLDRAM 3, QDR II+ / QDR II + Xtreme/QDR IV,

- **Administrador de Dispositivos Seguros:** La familia de dispositivos Intel® Stratix® 10 presenta un nuevo Administrador de dispositivos seguros (SDM) disponible en todas las densidades y variantes de familias de dispositivos. Sirviendo como el centro de comando central para todo el FPGA, el Administrador de dispositivos seguros controla las operaciones clave, como la configuración, la seguridad del dispositivo, las respuestas SEU y la administración de energía. El administrador de dispositivos seguros crea un sistema de administración seguro y unificado para todo el dispositivo, incluido el tejido FPGA, el sistema de procesador duro (HPS) en los dispositivo de sistema integrado en chip, los bloques de IP duros integrados y los bloques de E/S.

- **Sistema de procesador físico:** Los dispositivos del sistema integrado en chips de Intel® Stratix® 10 incluyen un sistema de procesador duro (hard processor, HPS) de próxima generación para ofrecer los dispositivo de sistema integrado en chip con el mayor desempeño y la mayor eficiencia energética de la industria. En el corazón del HPS se encuentra un clúster de procesador ARM* Cortex*-A53 de cuatro núcleos altamente eficiente. Este procesador está optimizado para un desempeño ultra alto por vatio, lo que reduce el consumo de energía hasta en un 50 % en comparación con los FPGA dispositivo de sistema integrado en chip de la generación anterior. Además, el HPS incluye una unidad de gestión de memoria del sistema, una unidad de coherencia de caché, un controlador de memoria duro y un conjunto completo de características de periféricos integrados.

Todo lo anterior conlleva a lo siguiente: a) Alto desempeño: La integración heterogénea proporciona un camino para integrar capacidades de interfaz de mayor ancho de banda para satisfacer las necesidades de los sistemas de 400 Gigabit a 1 Terabit; b) Menor consumo de energía: En comparación con los componentes discretos en una placa de circuito impreso, la integración heterogénea reduce la cantidad de energía consumida al impulsar interconexiones largas para ofrecer una solución general de menor consumo; c) Factor de forma más pequeño: Se integran componentes discretos en un solo paquete, para que el tamaño general de la solución se puede reducir significativamente, lo que incluye menos área de placa utilizada para el enrutamiento.

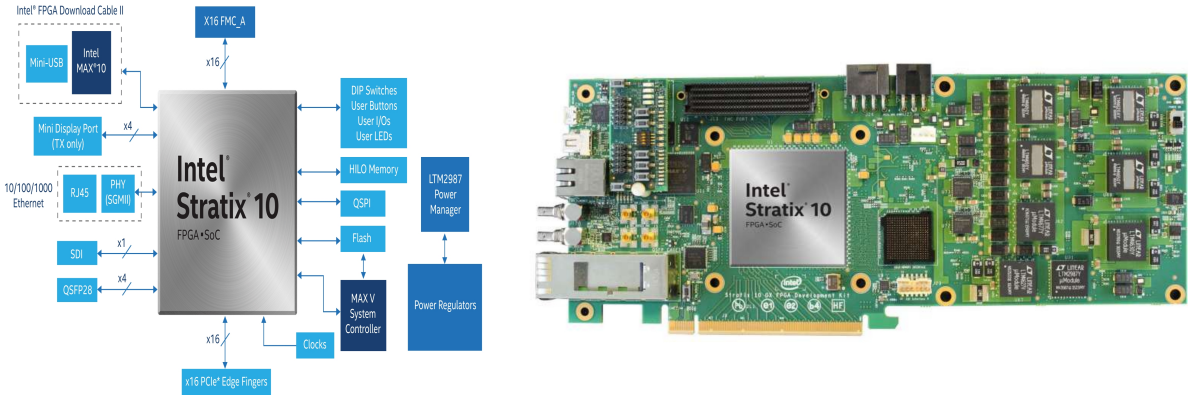


Figura B.3: Modelo general de la Arquitectura FPGA Intel® Stratix® 10GX 10M.
Fuente: www.intel.la.com

ANEXO C:
Código Fuente de la PAPI SkeletonCoRe en Lenguaje OpenCL C/C++

C.1 Código fuente secuencial en C/C++ de la Aplicación con Operadores de Procesamiento Digital de Imágenes (sin esqueletos ni OpenCL).

```
1  //*********************************************************************************************
2  //* Nombre del programa:  Cabecera del programa de prueba con operadores de procesamiento de imagenes
3  //*                       digitales como Sobel Edge Detection, Color to Grayscale Converter and
4  //*                       Interpolation for Rotating Image
5  //* Programador o autor:  Carlos Acosta-León
6  //* Función del programa: Cabecera con las librerías del programa.
7  //* Lenguaje de program:  Lenguaje C/C++
8  //* Fecha:                25 de Febrero de 2023
9  //* Motivo:               Programa fuente usado como ejemplo en Tesis Doctoral Ciens de la Comp
10 //*********************************************************************************************
11 // DECLARACION DE ENCABEZADOS DEL LENGUAJE C/C++
12 #include <stdio.h>   // Available in both C and C++
13 #include <chrono>
14 #include <typeinfo>
15 #include <stdlib.h>
16 #include <string.h> // Supported both in C and C++
17 #include <iostream> // Exclusive to C++
18 #include <vector>   // An exclusive feature of C++
19 #include <ctime>
20 #include <time.h>   // for clock_t, clock()
21 #include <cmath>
22 #include <math.h>
23 #include <iomanip>
24 #include <jpeglib.h>
25
26 //* SECCIÓN DE DECLARACIÓN DE ESPACIOS DE NOMBRES
27 using namespace std;
28 using namespace std::chrono; // Libreria de tiempo de alta precisión
29
30 //* SECCIÓN DE DECLARACIÓN DE CONSTANTES
31 #define SEC      1.0          // Time in seconds
32 #define MILLISEC 1000.0       // Time in milliseconds
33 #define MICROSEC 1000000.0    // Time in microseconds
34 #define NANOSEC  1000000000.0 // Time in nanoseconds
35
36 /* SECCIÓN DE DECLARACIÓN DE FUNCIONES Y PROCEDIMIENTOS EN C */
37 static inline int read_JPEG_file(char *filename, int *width, int *height, int *channels, unsigned
38     char *(image[]));
39 static inline void write_JPEG_file(const char *filename, int width, int height, int channels,
40     unsigned char image[], int quality);
41 static inline unsigned char *kernel_sobelEdgeDetection(unsigned char *image, int iWidth, int iHeight,
42     int channels);
43 static inline unsigned char *kernel_convertJPEGimageGrayscale(unsigned char *image, int width, int
44     height, int *channels);
45 static inline unsigned char *kernel_imageRotate(unsigned char *image, int width, int height,
46     int channels);
47 /*************************************************************/
48 /* API CON LA SECCIÓN DE DEFINICIONES DE FUNCIONES Y PROCEDIMIENTO EN C */
49 /*-----------------------------------------------------------*/
50 /*****************************************************************************
51 * Función: Read the JPEG image at `filename` as an array of bytes.
52 * Data is returned through the out pointers, while the return
53 * value indicates success or failure.
54 * NOTE - 1) if image is RGB, then the bytes are concatenated in R-G-B order
55 *        2) `image` should be freed by the user
56 *****************************************************************************/
57 static inline int read_JPEG_file(char *filename, int *width, int *height, int *channels, unsigned
58     char *(image[])) {
```

Código fuente B.17: Cabecera (header.hpp) del programa principal en el código fuente secuencial en C/C++ mostrado en **??**. Esta cabecera o "header" (.hpp) contiene las definiciones de variables, funciones y procedimiento usados en el referido programa principal (sin esqueletos ni OpenCL). Fuente: Elaborado por el autor.

```
59      /* Image input file Descriptor */
60      FILE *infile;
61
62      /* Opening the input image file */
63      if((infile = fopen(filename, "rb")) == NULL) {
64          fprintf(stderr, "can't open %s\n", filename);
65          return 0;
66      }
67
68      /* Getting the input image properties */
69      struct jpeg_error_mgr jerr;
70      struct jpeg_decompress_struct cinfo;
71      cinfo.err = jpeg_std_error(&jerr);
72      jpeg_create_decompress(&cinfo);
73      jpeg_stdio_src(&cinfo, infile);
74      (void) jpeg_read_header(&cinfo, TRUE);
75      (void) jpeg_start_decompress(&cinfo);
76
77      *width    = cinfo.output_width;
78      *height   = cinfo.output_height;
79      *channels = cinfo.num_components;
80
81      //* Printing the image dimensions
82      printf("Imput Image dimensions: Width = %d, Height = %d, Channels = %d\n", *width, *height, *channels);
83
84      /* Allocating memory space for the input image */
85      *image = (unsigned char*)malloc(*width * *height * *channels * sizeof(*image));
86      JSAMPROW rowptr[1];
87      int row_stride = *width * *channels;
88
89       while(cinfo.output_scanline < cinfo.output_height) {
90          rowptr[0] = *image + row_stride * cinfo.output_scanline;
91          jpeg_read_scanlines(&cinfo, rowptr, 1);
92       }
93
94      /* Finalizando la construcción de la imagen .jpg */
95      jpeg_finish_decompress(&cinfo);
96      jpeg_destroy_decompress(&cinfo); // Descomprimiendo imagen
97      fclose(infile);
98
99      return 1;
100 }
101 //
102 /*******************************************************************************
103 * Función: Writes the image in the specified file.
104 * NOTE - works with Grayscale or RGB modes only (based on number of channels)
105 *******************************************************************************/
106 static inline void write_JPEG_file(const char *filename, int width, int height, int channels,
107     unsigned char image[], int quality) {
108     /* Image output file Descriptor */
109     FILE *outfile;
110
111     /* Opening the output image file */
112     if ((outfile = fopen(filename, "wb")) == NULL) {
113         fprintf(stderr, "can't open %s\n", filename);
114         exit(1);
115     }
116
117     /* Construcción de las propiedades del formato .jpg del archivo de imagen de salida */
118     struct jpeg_error_mgr jerr;
```

(Cont. del Código fuente del Header B.17) Esta cabecera o "header" (.hpp) contiene las definiciones de variables, funciones y procedimiento usados en el referido programa principal en **??** (sin esqueletos ni OpenCL). Fuente: Elaborado por el autor.

```
118     struct jpeg_compress_struct cinfo;
119     cinfo.err = jpeg_std_error(&jerr);
120     jpeg_create_compress(&cinfo);
121     jpeg_stdio_dest(&cinfo,outfile);
122 
123     cinfo.image_width      = width;
124     cinfo.image_height     = height;
125     cinfo.input_components = channels;
126     cinfo.in_color_space   = channels == 1 ? JCS_GRAYSCALE : JCS_RGB;
127     jpeg_set_defaults(&cinfo);
128     jpeg_set_quality(&cinfo, quality, TRUE);
129 
130     jpeg_start_compress(&cinfo, TRUE); // Comprimiendo imagen .jpg de salida
131     JSAMPROW rowptr[1];
132     int row_stride = width * channels;
133 
134     while(cinfo.next_scanline < cinfo.image_height) {
135         rowptr[0] = & image[cinfo.next_scanline * row_stride];
136         jpeg_write_scanlines(&cinfo, rowptr, 1);
137     }
138 
139     /* Finalizando y liberando recursos */
140     jpeg_finish_compress(&cinfo);
141     fclose(outfile);
142     jpeg_destroy_compress(&cinfo);
143 }
144 //
145 /*********************************************************************************
146 * Función: Convierte una imagen a color en formato .jpeg o .jpg a una imagen en
147 * escala de 255 tonos de grises tambien en formato .jpg
148 *********************************************************************************/
149 static inline unsigned char *kernel_convertJPEGimageGrayscale(unsigned char *inImage, int width, int
150     height, int *channels) {
151     int           *imageAux;
152     unsigned char *outImage;
153 
154     imageAux = (int *)malloc(width * height * *channels * sizeof(inImage));
155     outImage = (unsigned char *)malloc(width * height * *channels * sizeof(inImage));
156 
157     /* En caso de imagen a color (3 canales RGB) se convierte a escala de grises (1 canal grises) */
158     if(*channels == 3) {
159         for (int i = 0; i < height; i++) {
160             for (int j = 0; j < width; j++) {
161                 imageAux[i*width+j] = (inImage[i*width*3+j*3] + inImage[i*width*3+j*3+1] +
162                     inImage[i*width*3+j*3+2])/3;
163             }
164         }
165         //printf("Procesando imagen No. %i:\n", i);
166 
167         // Se ha convertido una imagen de 3 canales a color RGB a 1 canal escala de 255 grises
168         *channels = 1;
169     }
170     // En caso de 1 canal para escala de 255 grises
171     else if(*channels == 1) {
172         for (int i = 0; i < height; i++) {
173             for (int j = 0; j < width; j++) {
174                 imageAux[i*width+j] = inImage[i*width+j];
175             }
176         }
177         //printf("Procesando imagen No. %i:\n", i);
```

(Cont. del Código fuente del Header B.17) Esta cabecera o “header” (.hpp) contiene las definiciones de variables, funciones y procedimiento usados en el referido programa principal en **??** (sin esqueletos ni OpenCL). Fuente: Elaborado por el autor.

```
178
179         // Se ha convertido una imagen de 3 canales a color RGB a 1 canal escala de 255 grises
180         *channels = 1;
181     }
182
183     // Se prepara y transfiere la imagen resultante
184     for (int i = 0; i < height; i++){
185         for (int j = 0; j < width; j++) {
186             outImage[i*width+j] = imageAux[i*width+j];
187         }
188     }
189
190     // Libera espacio de memoria reservado
191     free(imageAux);
192     // Retornando la imagen resultado
193     return outImage;
194 }
195 //
196 /*****************************************************************************
197 * Función que aplica el algoritmo Sobel para detección de borde de imágenes
198 *****************************************************************************/
199 static inline unsigned char *kernel_sobelEdgeDetection(unsigned char *image, int iWidth, int iHeight,
200     int channels) {
201     int *pImg, *pImg2;
202     unsigned char *imageAux;
203     int pixel_x;
204     int pixel_y;
205
206     // Reserva espacio de memoria a matrices unidimensionales
207     pImg  = (int *)malloc(iWidth* iHeight* channels* sizeof(image));
208     pImg2 = (int *)malloc(iWidth* iHeight* channels* sizeof(image));
209     imageAux = (unsigned char *)malloc(iWidth* iHeight* channels* sizeof(image));
210
211     // Copiando arreglo de imagen a uno auxiliar temporal
212     for (int i = 0; i < iHeight; i++){
213         for (int j = 0; j < iWidth; j++) {
214             pImg[i*iWidth+j] = image[i*iWidth+j];
215         }
216     }
217
218     // Aplicando el máscaras del algoritmo Sobel a la matriz de la imagen
219     // Gradiente Horizontal:
220     // Máscara G_x = +1, +2, +1
221     //                0,  0,  0
222     //               -1, -2, -1
223     /*float G_x[3][3] = { {-1, 0, +1},
224                           {-2, 0, +2},
225                           {-1, 0, +1}
226                           };*/
227     float G_x[3][3] = { {+1, +2, +1},
228                         { 0,  0,  0},
229                         {-1, -2, -1}
230                         };
231
232     // Gradiente Vertical:
233     // Máscara G_y = +1, 0, -1
234     //               +2, 0, -2
235     //               +1, 0, -1
236     /*float G_y[3][3] = { {-1, -2, -1},
237                           { 0,  0,  0},
238                           { 1,  2,  1}
```

(Cont. del Código fuente del Header B.17) Esta cabecera o "header" (.hpp) contiene las definiciones de variables, funciones y procedimiento usados en el referido programa principal en **??** (sin esqueletos ni OpenCL). Fuente: Elaborado por el autor.

```
238                         };*/
239     float G_y[3][3] = { {+1, 0, -1},
240                         {+2, 0, -2},
241                         {+1, 0, -1}
242                         };
243 
244     for (int x=1; x < iWidth-1; x++) {
245         for (int y=1; y < iHeight-1; y++) {
246             pixel_x = (G_x[0][0] * pImg[iWidth * (y-1) + (x-1)])
247                     + (G_x[0][1] * pImg[iWidth * (y-1) +  x   ])
248                     + (G_x[0][2] * pImg[iWidth * (y-1) + (x+1)])
249                     + (G_x[1][0] * pImg[iWidth *  y    + (x-1)])
250                     + (G_x[1][1] * pImg[iWidth *  y    +  x   ])
251                     + (G_x[1][2] * pImg[iWidth *  y    + (x+1)])
252                     + (G_x[2][0] * pImg[iWidth * (y+1) + (x-1)])
253                     + (G_x[2][1] * pImg[iWidth * (y+1) +  x   ])
254                     + (G_x[2][2] * pImg[iWidth * (y+1) + (x+1)]);
255 
256             pixel_y = (G_y[0][0] * pImg[iWidth * (y-1) + (x-1)])
257                     + (G_y[0][1] * pImg[iWidth * (y-1) +  x   ])
258                     + (G_y[0][2] * pImg[iWidth * (y-1) + (x+1)])
259                     + (G_y[1][0] * pImg[iWidth *  y    + (x-1)])
260                     + (G_y[1][1] * pImg[iWidth *  y    +  x   ])
261                     + (G_y[1][2] * pImg[iWidth *  y    + (x+1)])
262                     + (G_y[2][0] * pImg[iWidth * (y+1) + (x-1)])
263                     + (G_y[2][1] * pImg[iWidth * (y+1) +  x   ])
264                     + (G_y[2][2] * pImg[iWidth * (y+1) + (x+1)]);
265 
266             int val = (int)sqrt((pixel_x * pixel_x) + (pixel_y * pixel_y));
267             if(val < 0)   val = 0;
268             if(val > 255) val = 255;
269 
270             pImg2[iHeight * y + x] = val;
271         }
272     }
273 
274     // Se prepara y transfiere la imagen resultante
275     for (int i = 0; i < iHeight; i++){
276         for (int j = 0; j < iWidth; j++) {
277             imageAux[i*iWidth+j] = pImg2[i*iWidth+j];
278         }
279     }
280 
281     // Liberando espacio de memoria asigado a arreglos de imágenes
282     free(pImg);
283     free(pImg2);
284 
285     // Retornando la imagen resultado
286     return imageAux;
287 }
288 //
289 /*****************************************************************************
290 * Función: Interpola rotando o aumentando el tamaño de una imagen
291 * escala de 255 tonos de grises tambien en formato .jpg
292 *****************************************************************************/
293 static inline unsigned char *kernel_imageRotate(unsigned char *image, int iWidth, int
294     iHeight, int channels) {
295 
296     //* Reserva espacio de memoria para imagen auxiliar
297     unsigned char *imageA = (unsigned char*)malloc(iWidth * iHeight * channels * sizeof(image));
```

(Cont. del Código fuente del Header B.17) Esta cabecera o “header” (.hpp) contiene las definiciones de variables, funciones y procedimiento usados en el referido programa principal en **??** (sin esqueletos ni OpenCL). Fuente: Elaborado por el autor.

```
299     //* Inicalizan indices para recorrer la matriz
300     int columnas = iWidth;
301     int filas    = iHeight;
302 
303     int rotar = 90;
304     if(channels == 3) {
305         // Si la imagen de entrada es a color RGB (RED, GREEN, BLUE)
306         switch(rotar) {
307             case 90:
308                 // Rota la imagen hacia la derecha 90 grados (sin invertir)
309                 for(int i = 0; i < filas; i++) {
310                     for(int j = 0; j < columnas; j++) {
311                         imageA[((j+1)*(channels*iWidth))-((i+1)*channels)+0] =
312                             image[channels*(i*iWidth+j)+0]; // RED
313                         imageA[((j+1)*(channels*iWidth))-((i+1)*channels)+1] =
314                             image[channels*(i*iWidth+j)+1]; // GREEN
315                         imageA[((j+1)*(channels*iWidth))-((i+1)*channels)+2] =
316                             image[channels*(i*iWidth+j)+2]; // BLUE
317                     }
318                 }
319                 //puts("La imagen a color ha sido rotada 90 con éxito!\n");
320                 break;
321             case 180:
322                 //* Rota la imagen hacia abajo = 180 grados sentido del reloj
323                 for(int i = 0; i < filas; i++) {
324                     for(int j = 0; j < columnas; j++) {
325                         imageA[(iWidth-i)*(iWidth*channels)-(channels*(j+1))+0] =
326                             image[channels*(i*iWidth+j)+0];
327                         imageA[(iWidth-i)*(iWidth*channels)-(channels*(j+1))+1] =
328                             image[channels*(i*iWidth+j)+1];
329                         imageA[(iWidth-i)*(iWidth*channels)-(channels*(j+1))+2] =
330                             image[channels*(i*iWidth+j)+2];
331                     }
332                 }
333                 //puts("La imagen a color ha sido rotada 180 con éxito!\n");
334                 break;
335             case 270:
336                 //* Rota la imagen hacia la izquierda = 270 grados sentido del reloj
337                 for(int i = 0; i < filas; i++) {
338                     for(int j = 0; j < columnas; j++) {
339                         imageA[(iWidth-j-1)*(channels*iWidth)+(i*channels)+0] =
340                             image[channels*(i*iWidth+j)+0]; // RED
341                         imageA[(iWidth-j-1)*(channels*iWidth)+(i*channels)+1] =
342                             image[channels*(i*iWidth+j)+1]; // GREEN
343                         imageA[(iWidth-j-1)*(channels*iWidth)+(i*channels)+2] =
344                             image[channels*(i*iWidth+j)+2]; // BLUE
345                     }
346                 }
347                 //puts("La imagen a color ha sido rotada 270 con éxito!\n");
348                 break;
349             default:
350                 // Deja la imagen sin rotar, queda como la original
351                 for(int i = 0; i < filas; i++) {
352                     for(int j = 0; j < columnas; j++) {
353                         imageA[channels*(i*iWidth+j)+0] = image[channels*(i*iWidth+j)+0]; // RED
354                         imageA[channels*(i*iWidth+j)+1] = image[channels*(i*iWidth+j)+1]; // GREEN
355                         imageA[channels*(i*iWidth+j)+2] = image[channels*(i*iWidth+j)+2]; // BLUE
356                     }
357                 }
358                 //puts("No se ha rotado la imagen!\n");
```

(Cont. del Código fuente del Header B.17) Esta cabecera o “header” (.hpp) contiene las definiciones de variables, funciones y procedimiento usados en el referido programa principal en **??** (sin esqueletos ni OpenCL). Fuente: Elaborado por el autor.

```
359                 //puts("No se ha rotado la imagen!\n");
360             }
361         }
362         // Caso contrario, la imagen de entrada es en tonos de grises
363         else if(channels == 1) {
364             // Si la imagen de entrada es a color RGB (RED, GREEN, BLUE)
365             switch(rotar) {
366                 case 90:
367                     // Rota la imagen hacia la derecha 90 grados (sin invertir)
368                     for(int i = 0; i < filas; i++) {
369                         for(int j = 0; j < columnas; j++) {
370                             imageA[((j+1)*(channels*iWidth))-((i+1)*channels)+0] =
371                                 image[channels*(i*iWidth+j)+0]; // RED
372                             imageA[((j+1)*(channels*iWidth))-((i+1)*channels)+1] =
373                                 image[channels*(i*iWidth+j)+1]; // GREEN
374                             imageA[((j+1)*(channels*iWidth))-((i+1)*channels)+2] =
375                                 image[channels*(i*iWidth+j)+2]; // BLUE
376                         }
377                     }
378                     //puts("La imagen a color ha sido rotada 90 con éxito!\n");
379                     break;
380                 case 180:
381                     //* Rota la imagen hacia abajo = 180 grados sentido del reloj
382                     for(int i = 0; i < filas; i++) {
383                         for(int j = 0; j < columnas; j++) {
384                             imageA[(iWidth-i)*(iWidth*channels)-(channels*(j+1))+0] =
385                                 image[channels*(i*iWidth+j)+0];
386                             imageA[(iWidth-i)*(iWidth*channels)-(channels*(j+1))+1] =
387                                 image[channels*(i*iWidth+j)+1];
388                             imageA[(iWidth-i)*(iWidth*channels)-(channels*(j+1))+2] =
389                                 image[channels*(i*iWidth+j)+2];
390                         }
391                     }
392                     //puts("La imagen a color ha sido rotada 180 con éxito!\n");
393                     break;
394                 case 270:
395                     //* Rota la imagen hacia la izquierda = 270 grados sentido del reloj
396                     for(int i = 0; i < filas; i++) {
397                         for(int j = 0; j < columnas; j++) {
398                             imageA[(iWidth-j-1)*(channels*iWidth)+(i*channels)+0] =
399                                 image[channels*(i*iWidth+j)+0]; // RED
400                             imageA[(iWidth-j-1)*(channels*iWidth)+(i*channels)+1] =
401                                 image[channels*(i*iWidth+j)+1]; // GREEN
402                             imageA[(iWidth-j-1)*(channels*iWidth)+(i*channels)+2] =
403                                 image[channels*(i*iWidth+j)+2]; // BLUE
404                         }
405                     }
406                     //puts("La imagen a color ha sido rotada 270 con éxito!\n");
407                     break;
408                 default:
409                     // Deja la imagen sin rotar, queda como la original
410                     for(int i = 0; i < filas; i++) {
411                         for(int j = 0; j < columnas; j++) {
412                             imageA[channels*(i*iWidth+j)+0] = image[channels*(i*iWidth+j)+0]; // RED
413                             imageA[channels*(i*iWidth+j)+1] = image[channels*(i*iWidth+j)+1]; // GREEN
414                             imageA[channels*(i*iWidth+j)+2] = image[channels*(i*iWidth+j)+2]; // BLUE
415                         }
416                     }
417                     //puts("No se ha rotado la imagen!\n");
418             }
419         }
```

(Cont. del Código fuente del Header B.17) Esta cabecera o "header" (.hpp) contiene las definiciones de variables, funciones y procedimiento usados en el referido programa principal en **??** (sin esqueletos ni OpenCL). Fuente: Elaborado por el autor.

```
420
421     // Retornando la imagen resultado
422     return imageA;
423 }
424 //
425 /*****************************************************************************
426 * Clase que encapsula la gestión de la imagen según su formato jpg, jpeg, gif, bmp, etc.
427 *****************************************************************************/
428 namespace imagenes {
429     /* Clase Gestión de Imagenes */
430     class imagen {
431         public:
432             // Declaración de atributos públicos de la CLASE
433             int     imageType;
434             string  imageName;
435             char    *fileName;
436             int     width;
437             int     height;
438             int     RGBChannels;
439             int     GrayChannel;
440             int     Channels;
441             unsigned char *image;
442             unsigned char *ent_Imagen;         /* Contenedor de imagen Sobel de salida */
443             unsigned char *res_imageSobel;     /* Contenedor de imagen Sobel de salida */
444             unsigned char *res_imageGrayscale; /* Contenedor de imagen Grayscale de salida */
445             unsigned char *res_imageRotated;   /* Contenedor de imagen Interpolation de salida */
446
447             // Declaración de métodos públicos de la CLASE
448             void readInputImageJPG(char *filename_, unsigned char *(image[]));
449             void writeOutputImageJPG(const char *filename, int width, int height, int channels,
450                 unsigned char (image[]), int quality);
451
452         // Declaración de atributos privados de la CLASE
453         private:
454             //int   varX;
455             //string  varY;
456     };
457     //
458     /* Definición de métodos públicos de la CLASE */
459     /*****************************************************************************
460     * Método: Lee y obtiene las propiedades de la imagen en formato .jpg
461     *****************************************************************************/
462     void imagen::readInputImageJPG(char *filename_, unsigned char *(image[])) {
463         // Declaración de variables locales
464         int width_, height_, channels_;
465
466         // Wrapper de la funcion en C para la extracción de las propiedades de la imagen de entrada
467         read_JPEG_file(filename_, &width_, &height_, &channels_, image);
468
469         // Llenando los campos públicos del objeto IMAGEN
470         fileName    = filename_;
471         width       = width_;
472         height      = height_;
473         Channels    = channels_;
474         RGBChannels = channels_; // 3 Canales a color RGB
475         GrayChannel = channels_; // 1 Canal de grises
476
477         // Datos de la imagen de entrada
478         /*
479         cout << "Archivo imagen de entrada    : " << fileName << "\n";
```

(Cont. del Código fuente del Header B.17) Esta cabecera o “header” (.hpp) contiene las definiciones de variables, funciones y procedimiento usados en el referido programa principal en **??** (sin esqueletos ni OpenCL). Fuente: Elaborado por el autor.

```
480         cout << "Formato de la imagen         : " << "jpg" << "\n";
481         cout << "Dimensiones de imagen (Alto) : " << height << "\n";
482         cout << "Dimensiones de imagen (Ancho): " << width << "\n";
483         cout << "Canales RGB                  : " << Channels << "\n";
484         cout << "-----------------------------------------------" << "\n";
485         */
486
487         // Tiempo de lectura de la imagen de entrada
488         //cout << "The Input time is: " << (*t_exec/(CLOCKS_PER_SEC)*MILLISEC) << " milliseconds\n";
489         //printf("The Input time is: %0.4f milliseconds\n", (*t_exec/(CLOCKS_PER_SEC))*MILLISEC);
490     }
491     /*****************************************************************************
492     * Método: Escribe en un archivo de salida la imagen resultante  en formato .jpg
493     *****************************************************************************/
494     void imagen::writeOutputImageJPG(const char *filename, int width, int height, int channels,
495         unsigned char image[], int quality) {
496
497         // Wrapper de la funcion en C para grabar la imagen de salida
498         write_JPEG_file(filename, width, height, channels, image, quality);
499
500         // Tiempo de Escritura de la imagen de salida
501         //cout << "The Output time is: " << (*t_exec/(CLOCKS_PER_SEC)*MILLISEC) << " milliseconds\n";
502         //printf("The Output time is: %0.4f milliseconds\n", (*t_exec/(CLOCKS_PER_SEC))*MILLISEC);
503
504     }
505     /*****************************************************************************/
506     /* Clase Gestión de Operaciones de Imagenes */
507     class imageKernel {
508         public:
509             // Declaración de atributos públicos de la CLASE
510             /* Arreglos contenedores de imagenes de salida */
511             //unsigned char *ent_Imagen;      /* Contenedor de imagen Sobel de salida */
512             //unsigned char *res_imageSobel;      /* Contenedor de imagen Sobel de salida */
513             //unsigned char *res_imageGrayscale;  /* Contenedor de imagen Grayscale de salida */
514             //unsigned char *res_imageRotated;   /* Contenedor de imagen Interpolation de salida */
515
516             // Declaración de métodos públicos de la CLASE
517             inline void sobelEdgeDetection(unsigned char *inImage, unsigned char *(outImage[]), int
518                 iWidth, int iHeight, int channels);
519             inline void convertImageGrayscale(unsigned char *inImage, unsigned char *(outImage[]),
520                 int width, int height, int *channels);
521             inline void imageRotate_(unsigned char *inImage, unsigned char *(outImage[]), int
522                 width, int height, int channels);
523
524         // Declaración de atributos privados de la CLASE
525         private:
526             //int     var1;
527             //string  var2;
528     };
529     /* Definición de métodos públicos de la CLASE */
530     /*****************************************************************************
531     * Método: Convierte una imagen jpg a color en una imagen a escala de grises
532     *****************************************************************************/
533     inline void imageKernel::convertImageGrayscale(unsigned char *inImage, unsigned char
534         *(outImage[]), int width, int height, int *channels) {
535
536         // Wrapper de la funcion en C para convertir imagen a color a escala de grises
537         /*imageGrayscale_aux = kernel_convertJPEGimageGrayscale(inImage, width, height, channels);
538         *outImage = imageGrayscale_aux;*/
539         *outImage  = kernel_convertJPEGimageGrayscale(inImage, width, height, channels);
```

(Cont. del Código fuente del Header B.17) Esta cabecera o "header" (.hpp) contiene las definiciones de variables, funciones y procedimiento usados en el referido programa principal en **??** (sin esqueletos ni OpenCL). Fuente: Elaborado por el autor.

```
540         //*outImage = imageGrayscale_aux;
541
542         /* Se calcula o mide el tiempo total de procesamiento: Tiempo usado de CPU = fin - inicio */
543         /*
544         cout << "------------------------------------------------" << "\n";
545         cout << "Task 1: Execution Time for Converting to Grayscale:" << "\n";
546         cout << "Cantidad de imágenes procesadas: " << repetir << " de " << width << " bits :\n";
547         cout << "The elapsed cpu time is: " << (*t_exec/(CLOCKS_PER_SEC)) << " seconds\n";
548         cout << "The elapsed cpu time is: " << (*t_exec/(CLOCKS_PER_SEC)*MILLISEC) << " milliseconds\n";
549         cout << "------------------------------------------------" << "\n";
550         */
551     }
552     /*****************************************************************************
553     * Método: Aplica el algoritmo Sobel Edge Detection a imagenes en formato bitmap
554     *****************************************************************************/
555     inline void imageKernel::sobelEdgeDetection(unsigned char *inImage, unsigned char *(outImage[]),
556         int iWidth, int iHeight, int channels) {
557
558         // Wrapper de la funcion en C para detectar los bordes de la imagen
559         /*imageSobel_aux = kernel_sobelEdgeDetection(inImage, iWidth, iHeight, channels);
560         *outImage = imageSobel_aux;*/
561         *outImage = kernel_sobelEdgeDetection(inImage, iWidth, iHeight, channels);
562         //*outImage = imageSobel_aux;
563         /* Se calcula o mide el tiempo total de procesamiento: Tiempo usado de CPU = fin - inicio */
564         /*
565         cout << "------------------------------------------------" << "\n";
566         cout << "Task 2: Execution Time for Sobel Edge Detection:" << "\n";
567         cout << "Cantidad de imágenes procesadas: " << repetir << " de " << iWidth << " bits :\n";
568         cout << "The elapsed cpu time is: " << (*t_exec/(CLOCKS_PER_SEC)) << " seconds\n";
569         cout << "The elapsed cpu time is: " << (*t_exec/(CLOCKS_PER_SEC)*MILLISEC) << " milliseconds\n";
570         cout << "------------------------------------------------" << "\n";
571         */
572     }
573     /*****************************************************************************
574     * Método: Aplica el algoritmo Bicubic Interpolation a imagenes en formato bitmap
575     *****************************************************************************/
576     inline void imageKernel::imageRotate_(unsigned char *inImage, unsigned char *(outImage[]),
577         int width, int height, int channels) {
578
579     // Wrapper de la funcion en C para calcular la Interpolación de la imagen
580     /*imageRotated_aux = kernel_imageRotate(inImage, width, height, channels);
581     *outImage = imageRotated_aux;*/
582     *outImage = kernel_imageRotate(inImage, width, height, channels);
583     //*outImage = imageRotated_aux;
584
585     /* Se calcula o mide el tiempo total de procesamiento: Tiempo usado de CPU = fin - inicio */
586     /*
587     cout << "------------------------------------------------" << "\n";
588     cout << "Task 3: Execution Time for Bicubic Image Interpolation:" << "\n";
589     cout << "Cantidad de imágenes procesadas: " << repetir << " de " << width << " bits :\n";
590     cout << "The elapsed cpu time is: " << (*t_exec/(CLOCKS_PER_SEC)) << " seconds\n";
591     cout << "The elapsed cpu time is: " << (*t_exec/(CLOCKS_PER_SEC)*MILLISEC) << " milliseconds\n";
592     cout << "------------------------------------------------" << "\n";
593     */
594
595     }
596 }
597 //*******************************************************************
598 // DECLARACION DE ESPACIOS DE NOMBRES DE LAS CLASES ASOCIADAS A IMAGEN
599 //using namespace imagenes;
```

(Cont. del Código fuente del Header B.17) Esta cabecera o "header" (.hpp) contiene las definiciones de variables, funciones y procedimiento usados en el referido programa principal en **??** (sin esqueletos ni OpenCL). Fuente: Elaborado por el autor.

C.2 Código fuente en OpenCL C/C++ de la Interfaz de Programación de Aplicaciones Paralelas, PAPI SkeletonCoRe.

```
1
2  /*#############################################################################################
3  # Autor: Carlos Acosta (25/02/2022)
4  # Nombre del archivo: skeletoncore_api.hpp
5  # Función: Archivo Header en OpenCL C/C++ con las definiciones y declaraciones de todos los Métodos
6  #          (funciones), Atributos (datos), macros, clases, etc. del API SkeletonCoRe.
7  ##############################################################################################*/
8  // DECLARACION DE ENCABEZADOS DEL LENGUAJE C/C++
9  #include <stdio.h>   // Available in both C and C++
10 #include <iostream>
11 #include <chrono>
12 #include <typeinfo>
13 #include <stdlib.h>
14 #include <string.h> // Supported both in C and C++
15 #include <iostream> // Exclusive to C++
16 #include <vector>   // An exclusive feature of C++
17 #include <ctime>
18 #include <time.h>
19 // DECLARACION DEL ENCABEZADO DE OPENCL C/C++
20 #ifdef __APPLE__
21 #include <OpenCL/cl.h>
22 #else
23 //#define CL_HPP_TARGET_OPENCL_VERSION 220
24 //#define CL_USE_DEPRECATED_OPENCL_2_0_APIS
25 #define CL_HPP_TARGET_OPENCL_VERSION 210
26 #include <CL/cl2.hpp>
27 #endif
28
29 // Tamaño máximo del archivo kernel con el código de cómputo
30 #define MAX_SOURCE_SIZE (0x100000)
31
32 // DECLARACION DE ESPACIOS DE NOMBRES
33 using namespace std;
34 using namespace std::chrono; // Libreria de tiempo de alta precisión
35
36 //#########################################################################
37 //### ESPACIO DE NOMBRES DE LA CLASE API SKELETONCORE                   ###
38 //#########################################################################
39 namespace skeletoncore_api {
40     //#####################################################################
41     //### DECLARACIÓN DE ATRIBUTOS (VARIABLES) DEL API SKELETONCORE ###
42     //#####################################################################
43     //******* Estructura de Datos de la Partición *******
44     struct SkelcorePartition {
45         //******* Datos Públicos *******
46         string           partitionType;
47         FILE             *kernelFile;     // Parámetros para carga del Kernel desde el
48       char             *kernelSource;   // archivo "nombreArchKernel.cl"
49       size_t           kernelSize;
50       cl_program       program;         // Create program from kernel source
51         cl_kernel        kernel;          // Apuntador a la Tarea del Kernel de Cómputo de la Particion CPU
52         cl_device_id     deviceID = NULL; // Identificador del Dispositivo asociado a la particion
53         cl_context       context;         // Creación del contexto del CPU (Creating a context for CPU)
54         cl_mem           aMemObj;         // Creación de objetos de memoria (buffers) para contener datos
55         cl_mem           bMemObj;         // (Creating memory objects (buffers) to hold data).
56         cl_mem           cMemObj;
57         float            *datvector_A;    // Datos de Entrada y Salida de la Partición
58         float            *datvector_B;
59         float            *datvector_C;
60         cl_int           memVector_SIZE;  // Tamaño o dimensión de los vectores de datos y resultados
```

Código fuente B.18: Archivo de Encabezado en OpenCL C/C++ con las declaraciones y definiciones de las Clases con los Métodos de Propósito General y Esqueletos Algorítmicos del API de SkeletonCoRe. Fuente: Elaborado por el autor.

```
61          size_t           globalItemSize;
62        size_t           localItemSize;
63          cl_command_queue commandQueue;    // Creación de una cola de comandos para el dispositivo CPU
64          cl_int           ret = 0;         // Por descubrir
65      };
66      //#############################################################################
67      // ### DEFINICIÓN DE CLASES: METODOS DE PROPÓSITO GENERAL DEL API SKELETONCORE ###
68      //#############################################################################
69      /* Clase del API SkeletonCoRe */
70      class Skeletoncore_api {
71          /*** DECLARACIÓN DE ATRIBUTOS DEL API SKELETONCORE ***/
72          private:
73          // Datos e informacion general de configuracion de la plataforma y dispositivos OpenCL
74        //cl_platform_id platformID       = NULL;  // Identificador de la Plataforma
75          cl_device_id   deviceID         = NULL;  // Identificador del Dispositivo OpenCL CPU
76          cl_device_id   cpu_deviceID     = NULL;  // Identificador del Dispositivo OpenCL CPU
77          cl_device_id   gpu_deviceID     = NULL;  // Identificador del Dispositivo OpenCL GPU
78        cl_device_id   fpga_deviceID    = NULL;  // Identificador del Dispositivo OpenCL FPGA
79          cl_int         ret              = 0;     // Por descubrir
80          // Getting execution times information
81          float t_init_cpu,  t_end_cpu;
82          float t_init_gpu,  t_end_gpu;
83          float t_init_fpga, t_end_fpga;
84          public:
85          float t_cpu_exec;
86          float gpu_exec_time;
87          float fpga_exec_time;
88          //#############################################################################
89          // ### DECLARACIÓN DE METODOS DE PROPÓSITO GENERAL DEL API SKELETONCORE ###
90          //#############################################################################
91          // Métodos Públicos de la CLASE
92          public:
93                Skeletoncore_api();
94          void  init(); // Inicializa el API de SkeletonCore
95          void  terminate(SkelcorePartition &partition); // Sobrecarga del Método de Terminación
96          void  terminate(SkelcorePartition &partition1, SkelcorePartition &partition2);
97          void  terminate(SkelcorePartition &partition1, SkelcorePartition &partition2,
98                    SkelcorePartition &partition3);
99          void  readDataSize(SkelcorePartition &partition, int dataSIZE, int segmentSIZE);
100         void  readDataPartition(SkelcorePartition &partition, float* &vectorA, float* &vectorB, \
101                 float* &vectorC, int vector_size, int segment_size);
102         void  setupPartition(SkelcorePartition &partition, string Type, char *kernelName);
103         void  testResult(float *vec_A, float *vec_B, float *vec_C, int tamVec);
104         void  readData(float *vec_A, float *vec_B, float *vec_C, int tamVec);
105         void  writeResult(float *vec_A, float *vec_B, float *vec_C, int tamVec, char *outFileName);
106         void  printResult(float *vec_A, float *vec_B, float *vec_C, int tamVec);
107         //void  getExecTime(float t_init, float t_end);
108         void  getExecTime(long t_init, long t_end);
109         void  error_skelcore(int err_);
110         // Declaración de Métodos de Esqueletos Algoritmicos Paralelos del API
111         /* Parallel Skeleton con Sobrecarga */
112         void  parallelSkel(SkelcorePartition &partition);
113         void  parallelSkel(SkelcorePartition &partition1, SkelcorePartition &partition2);
114         void  parallelSkel(SkelcorePartition &partition1, SkelcorePartition &partition2,
115                   SkelcorePartition &partition3);
116         /* PipeSkeleton con Sobrecarga */
117         void  PipeSkeleton(SkelcorePartition &partition1);
118         void  PipeSkeleton(SkelcorePartition &partition1, SkelcorePartition &partition2);
119         void  PipeSkeleton(SkelcorePartition &partition1, SkelcorePartition &partition2,
120                   SkelcorePartition &partition3);
121         /* con TaskSkeleton con Sobrecarga*/
```

(Cont. Codigo fuente B.18) Archivo de Encabezado en OpneCL C/C++ con las declaraciones y definiciones de las Clases del API de SkeletonCoRe. Fuente: Elaborado por el autor.

```
122         void  TaskSkeleton(SkelcorePartition &partition1);
123         /* con MapReduceSkeleton con Sobrecarga*/
124         void  MapReduceSkeleton(SkelcorePartition &partition1);
125         void  SEQSkeleton(SkelcorePartition &partition1);
126         /*********************************************************************/
127         // Métodos Privados de la CLASE
128         private:
129         void    OpenCLplatformDiscovery(void);
130         void    getOpenCLPlatforms(cl_platform_id &platformID, cl_uint &retNumPlatforms );
131         void    platformDiscovery();
132         void    setupKernel(SkelcorePartition &partition, char *kernelName);
133         void    cpu_pipe(void);
134         void    gpu_pipe(void);
135         void    fpga_pipe(void);
136         void    task_pipe(void);
137         //******* FIN DECLARACION DE METODOS DE LA CLASE SKELETONCORE_API *******
138     };
139     /*********************************************************************/
140     /******* DEFINICIÓN DE DE METODOS DE LA CLASE SKELETONCORE_API *******/
141     /*********************************************************************/
142     /* Método Constructor del API SkeletonCoRe */
143     Skeletoncore_api::Skeletoncore_api() {
144         //std::cout << "-->  0.   Iniciando Ambiente del API SKELETONCORE...!\n";
145     }
146     /*************************************************************/
147     /* Method for Initialization and Discovering the platform and devices in the Heterogeneous System*/
148     void Skeletoncore_api::init(void) {
149         std::cout << "===>  ******* Beggining of the SKELETONCORE API Environment *******\n";
150         // Discovering the OpenCL Platform for Heterogeneous Computing
151         OpenCLplatformDiscovery();
152     }
153     /*************************************************************/
154     /*** Method for discovering the Opencl Platform and Devices   ***/
155     void Skeletoncore_api::OpenCLplatformDiscovery(void) {
156         char            queryBuffer[1024];
157         cl_int          clError;
158         cl_platform_id  platformID          = NULL; // Identificador de la Plataforma
159         const int       CL_DEVICE_TYPE_FPGA = 6;
160         cl_uint         retNumDevices       = 0;    // Cantidad de Dispositivos OpenCL en la Plataforma
161       cl_uint         retNumPlatforms     = 0;    // Número de Plataformas
162 
163         std::cout << "--> 01.   Discovering the Opencl Platform and Devices...!\n";
164         // Getting how many Platforms in the Computing System
165         getOpenCLPlatforms(platformID, retNumPlatforms);
166 
167       // Getting how many OpenCL devices in the Computing System (CPU, GPU and FPGA)
168         switch(retNumPlatforms) {
169             case 1:
170                 ret = clGetDeviceIDs(platformID, CL_DEVICE_TYPE_CPU, 1, &cpu_deviceID, &retNumDevices);
171                 break;
172             case 2:
173                 //*** Detecting the OpenCL Devices availables
174                 ret = clGetDeviceIDs(platformID, CL_DEVICE_TYPE_CPU, 1, &cpu_deviceID, &retNumDevices);
175                 //cout << "     CL_DEVICE_TYPE_CPU" << CL_DEVICE_TYPE_CPU << "\n";
176                 ret = clGetDeviceIDs(platformID, CL_DEVICE_TYPE_ALL, 1, &gpu_deviceID, &retNumDevices);
177                 //cout << "     CL_DEVICE_TYPE_CPU" << CL_DEVICE_TYPE_GPU << "\n";
178                 break;
179             case 3:
180                 cout << "     1.1b  Get Device ID for CPU, GPU and FPGA \n";
181                 ret = clGetDeviceIDs(platformID, CL_DEVICE_TYPE_CPU, 1, &cpu_deviceID, &retNumDevices);
182                 ret = clGetDeviceIDs(platformID, CL_DEVICE_TYPE_GPU, 1, &gpu_deviceID, &retNumDevices);
```

(Cont. Codigo fuente B.18) Archivo de Encabezado en OpneCL C/C++ con las declaraciones y definiciones de las Clases del API de SkeletonCoRe. Fuente: Elaborado por el autor.

```
183                 ret = clGetDeviceIDs(platformID, CL_DEVICE_TYPE_ACCELERATOR, 1, &fpga_deviceID,
184                             &retNumDevices);
185                 break;
186             default:
187                 // code to be executed if expression doesn't match any err_
188                 cout << "NO Get Devices ID...! \n";
189         }
190     }
191     /*************************************************************/
192     /*** Método que detecta la cantidad de Plataformas OpenCL disponibles ***/
193     //cl_uint Skeletoncore_api::getOpenCLPlatforms(cl_platform_id *platformID) {
194     void Skeletoncore_api::getOpenCLPlatforms(cl_platform_id &platformID, cl_uint &retNumPlatforms) {
195 
196         // Get the Number of Platforms available
197         // Note that the second parameter "platforms" is set to NULL. If this is NULL then this
198       // argument is ignored and the API returns the total number of OpenCL platforms available.
199         cl_int          clError;
200         char            queryBuffer[1024];
201       cl_platform_id  *platforms    = NULL;
202         cl_uint         num_platforms = 0;
203 
204       // Se reserva memoria para el arreglo de plataformas
205       platforms = (cl_platform_id *)malloc(sizeof(cl_platform_id)*num_platforms);
206         if((clGetPlatformIDs(0, NULL, &num_platforms)) == CL_SUCCESS) {
207             platforms = (cl_platform_id *)malloc(sizeof(cl_platform_id)*num_platforms);
208             if(clGetPlatformIDs(num_platforms, platforms, NULL) != CL_SUCCESS) {
209                 free(platforms);
210                 printf("Error in call to clGetPlatformIDs....\n Exiting");
211                 exit(0);
212             }
213         }
214         retNumPlatforms = num_platforms;
215         platformID = *platforms;
216         std::cout << "    01.1  OpenCL Platforms Available on this Heterogeneous System: " <<
217                   retNumPlatforms << "\n";
218         //************
219         if(num_platforms == 0) {
220             printf("No OpenCL Platforms Found ....\n Exiting");
221             exit(0);
222         }
223         else {
224             // We have obtained one platform here. Lets enumerate the devices available in this Platform.
225             for (cl_uint pIndex = 0; pIndex < num_platforms; pIndex++) {
226           cl_device_id    *devices;
227                 cl_uint          num_devices;
228 
229                 clError = clGetDeviceIDs (platforms[pIndex], CL_DEVICE_TYPE_ALL, 0, NULL, &num_devices);
230           printf("    01.2  Plataform ID=%d --> ", pIndex, num_devices);
231                 if (clError != CL_SUCCESS) {
232                     printf("Error Getting number of devices... Exiting\n ");
233                     exit(0);
234                 }
235                 // If successfull the num_devices contains the number of devices available in the platform
236                 // Now lets get all the device list. Before that we need to malloc devices
237           devices = (cl_device_id *)malloc(sizeof(cl_device_id) * num_devices);
238                 clError = clGetDeviceIDs (platforms[pIndex], CL_DEVICE_TYPE_ALL, num_devices, devices,\
239               &num_devices);
240                 if (clError != CL_SUCCESS) {
241                     printf("Error Getting number of devices... Exiting\n ");
242                     exit(0);
243                 }
```

(Cont. Codigo fuente B.18) Archivo de Encabezado en OpneCL C/C++ con las declaraciones y definiciones de las Clases del API de SkeletonCoRe. Fuente: Elaborado por el autor.

```
244
245                 for (cl_uint dIndex = 0; dIndex < num_devices; dIndex++) {
246             // Se obtiene el Nombre del Dispositivo de Cómputo OpenCL
247               clError = clGetDeviceInfo(devices[dIndex], CL_DEVICE_NAME, sizeof(queryBuffer),\
248                 &queryBuffer, NULL);
249                     std::cout << "OpenCL Divice detected: " << queryBuffer << "\n";
250               queryBuffer[0] = '\0';
251                 }
252             }
253         }
254     }
255     /*************************************************************/
256     /* Método para inicializar la Particion de Cómputo */
257     void Skeletoncore_api::setupPartition(SkelcorePartition &partition, string Type, char *kernelName) {
258         cl_platform_id  platformID          = NULL; // Identificador de la Plataforma
259         cl_uint         retNumPlatforms     = 0;    // Número de Plataformas
260         cl_uint         retNumDevices       = 0;    // Cantidad de Dispositivos OpenCL en la Plataforma
261
262         //*** Configurando los Parámetros de la Partición de Computación.
263         std::cout << "--> 02.   ==>Configurando la Partición de Cómputo del: " << Type << "\n";
264         if(Type == "CPU") {
265             partition.partitionType = Type; // Se crea e inicializa el objeto particion
266             std::cout << "    02.1  Creando su contexto de procesamiento" << "\n";
267             partition.context  = clCreateContext(NULL, 1, &cpu_deviceID, NULL, NULL,  &ret);
268             partition.deviceID = cpu_deviceID;
269         }
270         else {
271             if(Type == "GPU") {
272                 partition.partitionType = Type; // Se crea e inicializa el objeto particion
273                 std::cout << "    02.1  Creando su contexto de procesamiento" << "\n";
274                 partition.context  = clCreateContext(NULL, 1, &gpu_deviceID, NULL, NULL,  &ret);
275                 partition.deviceID = gpu_deviceID;
276
277             }
278             else {
279                 if(Type == "FPGA") {
280                     cout << "======> Es una Particion para FPGA \n";
281                 }
282                 else {
283                     cout << "======> Error dispositivo no reconocido! \n";
284                 }
285             }
286         }
287         //*** Creación de la cola de comandos para el Dispositivo (Creating a command-queue for CPU).
288         std::cout<<"    02.2  Creando la cola de comandos del " << partition.partitionType << "\n";
289       partition.commandQueue = clCreateCommandQueue(partition.context, partition.deviceID, 0, &ret);
290
291         //*** Creación de objetos de memoria (buffers) para contener datos de entrada/salida
292         //    (Creating memory objects (buffers) to hold data).
293       std::cout<<"    02.3  Creando objetos de memoria para datos\n";
294       partition.aMemObj = clCreateBuffer(partition.context, CL_MEM_READ_ONLY, \
295                                 partition.memVector_SIZE * sizeof(float), NULL, &ret); \
296       //std::cout<<"    II.4  Creando Buffer de memoria para Vector A\n";
297         partition.bMemObj = clCreateBuffer(partition.context, CL_MEM_READ_ONLY, \
298                                 partition.memVector_SIZE * sizeof(float), NULL, &ret);
299       //std::cout<<"    II.5  Creando Buffer de memoria para Vector B\n";
300         partition.cMemObj = clCreateBuffer(partition.context, CL_MEM_WRITE_ONLY, \
301                                 partition.memVector_SIZE * sizeof(float), NULL, &ret);
302         //std::cout<<"    II.6  Creando Buffer de memoria para Vector C\n";
303
304         //*** Copia de los datos de entrada en el dispositivo
```

(Cont. Codigo fuente B.18) Archivo de Encabezado en OpneCL C/C++ con las declaraciones y definiciones de las Clases del API de SkeletonCoRe. Fuente: Elaborado por el autor.

```
305         // (Copying the input data onto the device, Copy lists to memory buffers).
306       std::cout<<"    02.4  Copiando los datos al Dispositivo de Cómputo\n";
307       ret = clEnqueueWriteBuffer(partition.commandQueue, partition.aMemObj, CL_TRUE, 0, \
308                 partition.memVector_SIZE * sizeof(float), partition.datvector_A, 0, NULL, NULL);
309       ret = clEnqueueWriteBuffer(partition.commandQueue, partition.bMemObj, CL_TRUE, 0, \
310                 partition.memVector_SIZE * sizeof(float), partition.datvector_B, 0, NULL, NULL);
311         //std::cout<<"--> 03.   Configurando los Parámetros del Kernel\n";
312         setupKernel(partition, kernelName);
313     }
314     /*****************************************************************/
315     /* Métodos de Entrada/Salida del API SkeletonCoRe */
316     void Skeletoncore_api::setupKernel(SkelcorePartition &partition, char *kernelName) {
317         std::cout<<"    03.1  Configurando el Kernel asociado a la Partición...\n";
318       // Creación y compilación de un programa a partir del código fuente OpenCL C/C++ (Creating and
319       // compiling a program from the OpenCL C/C++ source code).
320       std::cout<<"    03.2  Creando y compilando el programa y Kernel OpenCL!\n";
321         partition.kernelFile = fopen(kernelName, "r");
322       if (!partition.kernelFile) {
323         fprintf(stderr, "No file named vecAddKernel.cl was found\n");
324         exit(-1);
325       }
326       partition.kernelSource = (char*)malloc(MAX_SOURCE_SIZE);
327       partition.kernelSize   = fread(partition.kernelSource, 1, MAX_SOURCE_SIZE, partition.kernelFile);
328       fclose(partition.kernelFile);
329       // Create program from kernel source
330       partition.program = clCreateProgramWithSource(partition.context, 1, \
331                 (const char **)&partition.kernelSource, (const size_t *)&partition.kernelSize, &ret);
332       // Build program
333       ret = clBuildProgram(partition.program, 1, &partition.deviceID, NULL, NULL, NULL);
334       // Extracción de Kernels del programa (Extracting the kernel from the program, Create kernel).
335       std::cout<<"    03.2  Extracción de Kernels de programa OpenCL\n";
336       partition.kernel = clCreateKernel(partition.program, "addVectors", &ret);
337       // Set arguments for kernel
338       ret = clSetKernelArg(partition.kernel, 0, sizeof(cl_mem), (void *)&partition.aMemObj);
339       ret = clSetKernelArg(partition.kernel, 1, sizeof(cl_mem), (void *)&partition.bMemObj);
340       ret = clSetKernelArg(partition.kernel, 2, sizeof(cl_mem), (void *)&partition.cMemObj);
341     }
342   /*****************************************************************/
343     /*** Método para Comprobar la Correctitud de los Resultados ***/
344     void Skeletoncore_api::testResult(float *vec_A, float *vec_B, float *vec_C, int tamVec) {
345         int i = 0;
346       for (i = 0; i < tamVec; ++i) {
347         if (vec_C[i] != (vec_A[i] + vec_B[i])) {
348           std::cout << "    04.2  ===> ...FALLÓ VERIFICACION DE RESULTADOS!\n";
349           break;
350         }
351       }
352       if (i == tamVec) {
353         std::cout << "    04.2  ===> Resultados verificados y correctos!\n";
354       }
355     }
356     /*****************************************************************/
357     /*** Función que graba en un archivo los resultados del procesamiento ***/
358     void Skeletoncore_api::writeResult(float *vec_A, float *vec_B, float *vec_C, int tamVec,
359         char *outFileName) {
360       // Declaración de variables locales
361         FILE *archivoSal;
362
363         archivoSal = fopen(outFileName, "w+"); // Nombre FILE y modo escritura: "w"
364         if(archivoSal==NULL) {
365             printf("Error creando el archivo de salida %s:\n", outFileName);
```

(Cont. Codigo fuente B.18) Archivo de Encabezado en OpneCL C/C++ con las declaraciones y definiciones de las Clases del API de SkeletonCoRe. Fuente: Elaborado por el autor.

```
366             exit(1);
367         }
368       // Grabando vectores en archivo de salida
369       for (int i = 0; i < tamVec; ++i) {
370         fprintf(archivoSal, "%f + %f = %f\n", vec_A[i], vec_B[i], vec_C[i]);
371       }
372         fclose(archivoSal); // Cerrando el archivo de salida
373       std::cout << "    04.3  ===> RESULTADOS GRABADOS EN: " << outFileName << "\n";
374     }
375     /*************************************************************/
376     /*** Print results on Display ***/
377     void Skeletoncore_api::printResult(float *vec_A, float *vec_B, float *vec_C, int tamVec) {
378       for (int i = 0; i < tamVec; ++i) {
379         printf("%f + %f = %f\n", vec_A[i], vec_B[i], vec_C[i]);
380       }
381     }
382     /*****************************************************************/
383     /*** Get the execution time from processing ***/
384     //void Skeletoncore_api::getExecTime(float t_init, float t_end) {
385     void Skeletoncore_api::getExecTime(long t_init, long t_end) {
386         // Calculate time in more precise durations (picoseconds...)
387
388         const long microseconds = 1000000l;       // 10^-6  seconds
389         const long nanosecondss = 1000000000l;    // 10^-9  seconds
390         const long picoseconds  = 10000000000000l; // 10^-12 seconds
391
392         std::cout << "-----------> Total Execution time is:" << (t_end-t_init) << "\n";
393     }
394     /*** Get the error messages from SkeletonCore API ***/
395     void  error_skelcore(int err_) {
396         err_ = 0;
397         switch (err_) {
398             case 1:
399                 cout << "SkeletonCore Error! (1): Message 1";
400                 break;
401             case 2:
402                 cout << "SkeletonCore Error! (2): Message 2";
403                 break;
404             case 3:
405                 cout << "SkeletonCore Error! (3): Message 3";
406                 break;
407             case 4:
408                 cout << "SkeletonCore Error! (4): Message 4";
409                 break;
410             case 5:
411                 cout << "SkeletonCore Error! (5): Message 5";
412                 break;
413             case 6:
414                 cout << "SkeletonCore Error! (6): Message 6";
415                 break;
416             case 7:
417                 cout << "SkeletonCore Error! (7): Message 7";
418                 break;
419             default:
420                 // Code to be executed if expression doesn't match any err_
421                 cout << "NO SkeletonCore Error! (0): Message 0";
422         }
423     }
424     /******* SObrecarga del Método *****************************************************/
425     /* Método para la Liberación de los recursos OpenCL (Releasing the OpenCL resources) */
426     void Skeletoncore_api::terminate(SkelcorePartition &partition1) {
```

(Cont. Codigo fuente B.18) Archivo de Encabezado en OpenCL C/C++ con las declaraciones y definiciones de las Clases con los Métodos de Propósito General y Esqueletos Algorítmicos del API de SkeletonCoRe. Fuente: Elaborado por el autor.

```
427       //std::cout<<"-->  6.   Se Liberan los recursos OpenCL asignados a la(s)
428       //      Particion(es) de Cómputo!\n";
429         // Clean up, release memory.
430         partition1.ret = clFlush(partition1.commandQueue);
431       partition1.ret = clFinish(partition1.commandQueue);
432       partition1.ret = clReleaseCommandQueue(partition1.commandQueue);
433       partition1.ret = clReleaseKernel(partition1.kernel);
434       partition1.ret = clReleaseProgram(partition1.program);
435       partition1.ret = clReleaseMemObject(partition1.aMemObj);
436       partition1.ret = clReleaseMemObject(partition1.bMemObj);
437       partition1.ret = clReleaseMemObject(partition1.cMemObj);
438       partition1.ret = clReleaseContext(partition1.context);
439         free(partition1.datvector_A);
440       free(partition1.datvector_B);
441       free(partition1.datvector_C);
442         //std::cout<<"-->  7.   Cerrando el API, Culminó con éxito el programa!\n";
443         std::cout << "===>  ******* Ending of the SKELETONCORE API Environment *******\n";
444     }
445     /* Método para la Liberación de los recursos OpenCL (Releasing the OpenCL resources) */
446     void Skeletoncore_api::terminate(SkelcorePartition &partition1, SkelcorePartition &partition2) {
447       //std::cout<<"-->  6.   Se Liberan los recursos OpenCL asignados a la(s) Particion(es)
448               // de Cómputo!\n";
449         // Clean up, release memory for CPU.
450         partition1.ret = clFlush(partition1.commandQueue);
451       partition1.ret = clFinish(partition1.commandQueue);
452       partition1.ret = clReleaseCommandQueue(partition1.commandQueue);
453       partition1.ret = clReleaseKernel(partition1.kernel);
454       partition1.ret = clReleaseProgram(partition1.program);
455       partition1.ret = clReleaseMemObject(partition1.aMemObj);
456       partition1.ret = clReleaseMemObject(partition1.bMemObj);
457       partition1.ret = clReleaseMemObject(partition1.cMemObj);
458       partition1.ret = clReleaseContext(partition1.context);
459         free(partition1.datvector_A);
460       free(partition1.datvector_B);
461       free(partition1.datvector_C);
462
463         // Clean up, release memory for GPU.
464         partition2.ret = clFlush(partition2.commandQueue);
465       partition2.ret = clFinish(partition2.commandQueue);
466       partition2.ret = clReleaseCommandQueue(partition2.commandQueue);
467       partition2.ret = clReleaseKernel(partition2.kernel);
468       partition2.ret = clReleaseProgram(partition2.program);
469       partition2.ret = clReleaseMemObject(partition2.aMemObj);
470       partition2.ret = clReleaseMemObject(partition2.bMemObj);
471       partition2.ret = clReleaseMemObject(partition2.cMemObj);
472       partition2.ret = clReleaseContext(partition2.context);
473         free(partition2.datvector_A);
474       free(partition2.datvector_B);
475       free(partition2.datvector_C);
476         //std::cout<<"-->  7.   Cerrando el API, Culminó con éxito el programa!\n";
477         std::cout << "===>  ******* Ending of the SKELETONCORE API Environment *******\n";
478     }
479     /* Método para la Liberación de los recursos OpenCL (Releasing the OpenCL resources) */
480     void Skeletoncore_api::terminate(SkelcorePartition &partition1, SkelcorePartition &partition2,
481           SkelcorePartition &partition3) {
482       //std::cout<<"-->  6.   Se Liberan los recursos OpenCL asignados a la(s) Particion(es)
483       //      de Cómputo!\n";
484         // Clean up, release memory for CPU.
485         partition1.ret = clFlush(partition1.commandQueue);
486       partition1.ret = clFinish(partition1.commandQueue);
487       partition1.ret = clReleaseCommandQueue(partition1.commandQueue);
```

(Cont. Codigo fuente B.18) Archivo de Encabezado en OpenCL C/C++ con las declaraciones y definiciones de las Clases con los Métodos de Propósito General y Esqueletos Algorítmicos del API de SkeletonCoRe. Fuente: Elaborado por el autor.

```
488       partition1.ret = clReleaseKernel(partition1.kernel);
489       partition1.ret = clReleaseProgram(partition1.program);
490       partition1.ret = clReleaseMemObject(partition1.aMemObj);
491       partition1.ret = clReleaseMemObject(partition1.bMemObj);
492       partition1.ret = clReleaseMemObject(partition1.cMemObj);
493       partition1.ret = clReleaseContext(partition1.context);
494         free(partition1.datvector_A);
495       free(partition1.datvector_B);
496       free(partition1.datvector_C);
497
498         // Clean up, release memory for GPU.
499         partition2.ret = clFlush(partition2.commandQueue);
500       partition2.ret = clFinish(partition2.commandQueue);
501       partition2.ret = clReleaseCommandQueue(partition2.commandQueue);
502       partition2.ret = clReleaseKernel(partition2.kernel);
503       partition2.ret = clReleaseProgram(partition2.program);
504       partition2.ret = clReleaseMemObject(partition2.aMemObj);
505       partition2.ret = clReleaseMemObject(partition2.bMemObj);
506       partition2.ret = clReleaseMemObject(partition2.cMemObj);
507       partition2.ret = clReleaseContext(partition2.context);
508         free(partition2.datvector_A);
509       free(partition2.datvector_B);
510       free(partition2.datvector_C);
511
512         // Clean up, release memory for FPGA.
513         partition3.ret = clFlush(partition3.commandQueue);
514       partition3.ret = clFinish(partition3.commandQueue);
515       partition3.ret = clReleaseCommandQueue(partition3.commandQueue);
516       partition3.ret = clReleaseKernel(partition3.kernel);
517       partition3.ret = clReleaseProgram(partition3.program);
518       partition3.ret = clReleaseMemObject(partition3.aMemObj);
519       partition3.ret = clReleaseMemObject(partition3.bMemObj);
520       partition3.ret = clReleaseMemObject(partition3.cMemObj);
521       partition3.ret = clReleaseContext(partition3.context);
522         free(partition3.datvector_A);
523       free(partition3.datvector_B);
524       free(partition3.datvector_C);
525         //std::cout<<"-->  7.   Cerrando el API, Culminó con éxito el programa!\n";
526         std::cout << "===>  ******* Ending of the SKELETONCORE API Environment *******\n";
527     }
528     /*************************************************************/
529     /*** Método que lee el tamaño de datos y granularidad de segmentos del dominio de datos ***/
530     void Skeletoncore_api::readDataSize(SkelcorePartition &partition, int dataSIZE, int segmentSIZE) {
531         // globalItemSize has to be a multiple of localItemSize. 1024/64 = 16
532         partition.globalItemSize = dataSIZE;
533       partition.localItemSize  = segmentSIZE;
534     }
535     /*************************************************************/
536     /* Métodos para la Creación de Contenedores de Datos Vector/Matriz */
537     void Skeletoncore_api::readDataPartition(SkelcorePartition &partition, float* &vectorA,
538         float* &vectorB, float* &vectorC, int vector_size, int segment_size) {
539         std::cout << "-->  2.   Leyendo Vectores de Datos de la Carga de Trabajo a Procesar...!\n";
540         // globalItemSize has to be a multiple of localItemSize. 1024/64 = 16
541         partition.globalItemSize = vector_size;
542       partition.localItemSize  = segment_size;
543
544         partition.memVector_SIZE = vector_size;
545         vectorA = (float*)malloc(sizeof(float)*vector_size); // Reserva memoria para Vector A
546         partition.datvector_A = vectorA;
547         vectorB = (float*)malloc(sizeof(float)*vector_size); // Reserva memoria para Vector B
548         partition.datvector_B = vectorB;
```

(Cont. Codigo fuente B.18) Archivo de Encabezado en OpenCL C/C++ con las declaraciones y definiciones de las Clases con los Métodos de Propósito General y Esqueletos Algorítmicos del API de SkeletonCoRe. Fuente: Elaborado por el autor.

```
549         vectorC = (float*)malloc(sizeof(float)*vector_size); // Reserva memoria para Vector C
550         partition.datvector_C = vectorC;
551         // Inicilizando los vectores de entrada A y B.
552       int i = 0;
553       for (i = 0; i < vector_size; ++i) {
554         partition.datvector_A[i] = i+1;
555         partition.datvector_B[i] = (i+1)*2;
556       }
557     }
558     /*****************************************************************/
559     /*** Método que detecta las plataformas y sus dispositivos de cómputo OpenCL disponibles ***/
560     void Skeletoncore_api::platformDiscovery(void) {
561         // DECLARACION DE VARIABLES LOCALES
562         char   queryBuffer[1024];
563         cl_int clError;
564 
565         // Get the Number of Platforms available
566         // Note that the second parameter "platforms" is set to NULL. If this is NULL then this
567       // argument is ignored and the API returns the total number of OpenCL platforms available.
568       cl_platform_id *platforms     = NULL;
569         cl_uint         num_platforms = 0;
570 
571       // Se reserva memoria para el arreglo de plataformas
572       platforms = (cl_platform_id *)malloc(sizeof(cl_platform_id)*num_platforms);
573         if((clGetPlatformIDs(0, NULL, &num_platforms)) == CL_SUCCESS) {
574             printf("     1.1  Número de Plataformas disponibles en el sistema heterogéneo: %d\n", \
575                 num_platforms);
576             platforms = (cl_platform_id *)malloc(sizeof(cl_platform_id)*num_platforms);
577 
578             if(clGetPlatformIDs(num_platforms, platforms, NULL) != CL_SUCCESS) {
579                 free(platforms);
580                 printf("Error in call to clGetPlatformIDs....\n Exiting");
581                 exit(0);
582             }
583         }
584         if(num_platforms == 0) {
585             printf("No OpenCL Platforms Found ....\n Exiting");
586             exit(0);
587         }
588         else {
589             // We have obtained one platform here. Lets enumerate the devices available in this Platform.
590             for (cl_uint pIndex = 0; pIndex < num_platforms; pIndex++) {
591           cl_device_id    *devices;
592                 cl_uint          num_devices;
593 
594                 clError = clGetDeviceIDs (platforms[pIndex], CL_DEVICE_TYPE_ALL, 0, NULL, &num_devices);
595           printf("     1.2  Plataforma ID(%d) con %d ", pIndex, num_devices);
596                 if (clError != CL_SUCCESS) {
597                     printf("Error Getting number of devices... Exiting\n ");
598                     exit(0);
599                 }
600                 // If successfull the num_devices contains the number of devices available in the platform
601                 // Now lets get all the device list. Before that we need to malloc devices
602           devices = (cl_device_id *)malloc(sizeof(cl_device_id) * num_devices);
603                 clError = clGetDeviceIDs (platforms[pIndex], CL_DEVICE_TYPE_ALL, num_devices, devices,\
604               &num_devices);
605                 if (clError != CL_SUCCESS) {
606                     printf("Error Getting number of devices... Exiting\n ");
607                     exit(0);
608                 }
```

(Cont. Codigo fuente B.18) Archivo de Encabezado en OpenCL C/C++ con las declaraciones y definiciones de las Clases con los Métodos de Propósito General y Esqueletos Algorítmicos del API de SkeletonCoRe. Fuente: Elaborado por el autor.

```
610                 for (cl_uint dIndex = 0; dIndex < num_devices; dIndex++) {
611               // Se obtiene el Nombre del Dispositivo de Cómputo OpenCL
612                 clError = clGetDeviceInfo(devices[dIndex], CL_DEVICE_NAME, sizeof(queryBuffer),\
613                   &queryBuffer, NULL);
614               printf("Dispositivo(s) OpenCL (ID %d): %s ", dIndex, queryBuffer); //CL_DEVICE_NAME
615                 queryBuffer[0] = '\0';
616               clError = clGetDeviceInfo(devices[dIndex], CL_DEVICE_VERSION, sizeof(queryBuffer),\
617                   &queryBuffer, NULL);
618               printf("--> (Versión %s)\n", queryBuffer); //CL_DEVICE_VERSION
619                 queryBuffer[0] = '\0';
620                   }
621               }
622           }
623       }
624       /*********************************************************************/
625       /******* DEFINICIÓN DE ESQUELETOS ALGORITMICOS RECONFIGURABLES *******/
626       /*********************************************************************/
627       //******* Sobrecarga del Esqueleto TaskSkeleton **************************
628       /*** Skeleton for Master/Slave Parallel Processing ***/
629       void Skeletoncore_api::TaskSkeleton(SkelcorePartition &partition1) {
630         std::cout << "--> 04.  Executing Parallel Skeleton \n";
631           // Execute the kernel
632           // So, globalItemSize has to be a multiple of localItemSize. 1024/64 = 16
633         ret = clEnqueueNDRangeKernel(partition1.commandQueue, partition1.kernel, 1, NULL, \
634                   &partition1.globalItemSize, &partition1.localItemSize, 0, NULL, NULL);
635           ret = clEnqueueNDRangeKernel(partition1.commandQueue, partition1.kernel, 1, NULL, \
636                   &dataSIZE, &segmentSIZE, 0, NULL, NULL);
637
638         // Copia de los datos de salida al Host (Copying output data back to the Host).
639         std::cout<<"-->  5.   Copiando resultados de salida hacia el Host o Anfitrión!\n";
640         // Read from device back to host.
641         ret = clEnqueueReadBuffer(partition1.commandQueue, partition1.cMemObj, CL_TRUE, 0, \
642                   partition1.memVector_SIZE * sizeof(float), partition1.datvector_C, 0, NULL, NULL);
643       }
644       /*** Skeleton for Master/Slave Parallel Processing ***/
645       void Skeletoncore_api::TaskSkeleton(SkelcorePartition &partition1, SkelcorePartition &partition2) {
646         std::cout << "--> 04.  Executing Parallel Skeleton \n";
647           // Execute the kernel CPU
648           // So, globalItemSize has to be a multiple of localItemSize. 1024/64 = 16
649         ret = clEnqueueNDRangeKernel(partition1.commandQueue, partition1.kernel, 1, NULL, \
650                   &partition1.globalItemSize, &partition1.localItemSize, 0, NULL, NULL);
651           //ret = clEnqueueNDRangeKernel(partition1.commandQueue, partition1.kernel, 1, NULL, \
652                   &dataSIZE, &segmentSIZE, 0, NULL, NULL);
653
654         // Copia de los datos de salida al Host (Copying output data back to the Host).
655         std::cout<<"-->  5.   Copiando resultados de salida hacia el Host o Anfitrión!\n";
656         // Read from device back to host.
657         ret = clEnqueueReadBuffer(partition1.commandQueue, partition1.cMemObj, CL_TRUE, 0, \
658                   partition1.memVector_SIZE * sizeof(float), partition1.datvector_C, 0, NULL, NULL);
659
660         std::cout << "--> 04.  Executing Parallel Skeleton \n";
661           // Execute the kernel GPU
662           // So, globalItemSize has to be a multiple of localItemSize. 1024/64 = 16
663         ret = clEnqueueNDRangeKernel(partition2.commandQueue, partition2.kernel, 1, NULL, \
664                   &partition2.globalItemSize, &partition2.localItemSize, 0, NULL, NULL);
665           //ret = clEnqueueNDRangeKernel(partition1.commandQueue, partition1.kernel, 1, NULL, \
666                   &dataSIZE, &segmentSIZE, 0, NULL, NULL);
667
668         // Copia de los datos de salida al Host (Copying output data back to the Host).
669         std::cout<<"-->  5.   Copiando resultados de salida hacia el Host o Anfitrión!\n";
670         // Read from device back to host.
```

(Cont. Codigo fuente B.18) Archivo de Encabezado en OpenCL C/C++ con las declaraciones y definiciones de las Clases con los Métodos de Propósito General y Esqueletos Algorítmicos del API de SkeletonCoRe. Fuente: Elaborado por el autor.

```
671       ret = clEnqueueReadBuffer(partition2.commandQueue, partition2.cMemObj, CL_TRUE, 0, \
672                 partition2.memVector_SIZE * sizeof(float), partition2.datvector_C, 0, NULL, NULL);
673     }
674     /*** Skeleton for Master/Slave Parallel Processing ***/
675     void Skeletoncore_api::TaskSkeleton(SkelcorePartition &partition1, SkelcorePartition &partition2, \
676     SkelcorePartition &partition3) {
677       std::cout << "--> 04.  Executing Parallel Skeleton \n";
678         // Execute the kernel CPU
679         // So, globalItemSize has to be a multiple of localItemSize. 1024/64 = 16
680       ret = clEnqueueNDRangeKernel(partition1.commandQueue, partition1.kernel, 1, NULL, \
681                 &partition1.globalItemSize, &partition1.localItemSize, 0, NULL, NULL);
682         //ret = clEnqueueNDRangeKernel(partition1.commandQueue, partition1.kernel, 1, NULL, \
683                 &dataSIZE, &segmentSIZE, 0, NULL, NULL);
684
685       // Copia de los datos de salida al Host (Copying output data back to the Host).
686       std::cout<<"-->  5.   Copiando resultados de salida hacia el Host o Anfitrión!\n";
687       // Read from device back to host.
688       ret = clEnqueueReadBuffer(partition1.commandQueue, partition1.cMemObj, CL_TRUE, 0, \
689                 partition1.memVector_SIZE * sizeof(float), partition1.datvector_C, 0, NULL, NULL);
690
691       std::cout << "--> 04.  Executing Parallel Skeleton \n";
692         // Execute the kernel GPU
693         // So, globalItemSize has to be a multiple of localItemSize. 1024/64 = 16
694       ret = clEnqueueNDRangeKernel(partition2.commandQueue, partition2.kernel, 1, NULL, \
695                 &partition2.globalItemSize, &partition2.localItemSize, 0, NULL, NULL);
696         //ret = clEnqueueNDRangeKernel(partition1.commandQueue, partition1.kernel, 1, NULL, \
697                 &dataSIZE, &segmentSIZE, 0, NULL, NULL);
698
699       // Copia de los datos de salida al Host (Copying output data back to the Host).
700       std::cout<<"-->  5.   Copiando resultados de salida hacia el Host o Anfitrión!\n";
701       // Read from device back to host.
702       ret = clEnqueueReadBuffer(partition2.commandQueue, partition2.cMemObj, CL_TRUE, 0, \
703                 partition2.memVector_SIZE * sizeof(float), partition2.datvector_C, 0, NULL, NULL);
704
705     // Copia de los datos de salida al Host (Copying output data back to the Host).
706     std::cout << "--> 04.  Executing Parallel Skeleton \n";
707         // Execute the kernel FPGA
708         // So, globalItemSize has to be a multiple of localItemSize. 1024/64 = 16
709       ret = clEnqueueNDRangeKernel(partition3.commandQueue, partition3.kernel, 1, NULL, \
710                 &partition3.globalItemSize, &partition3.localItemSize, 0, NULL, NULL);
711         //ret = clEnqueueNDRangeKernel(partition1.commandQueue, partition1.kernel, 1, NULL, \
712                 &dataSIZE, &segmentSIZE, 0, NULL, NULL);
713
714       // Copia de los datos de salida al Host (Copying output data back to the Host).
715       std::cout<<"-->  5.   Copiando resultados de salida hacia el Host o Anfitrión!\n";
716       // Read from device back to host.
717       ret = clEnqueueReadBuffer(partition3.commandQueue, partition3.cMemObj, CL_TRUE, 0, \
718                 partition3.memVector_SIZE * sizeof(float), partition3.datvector_C, 0, NULL, NULL);
719     }
720     //*************************************************************************
721     /*** Skeleton for Pipeline Parallel Processing (Sobrecarga del Esqueleto) ***/
722     void Skeletoncore_api::PipeSkeleton(SkelcorePartition &partition1) {
723       std::cout << "--> 04.   Executing Parallel Pipeline Skeleton... ";
724         // Execute the kernel CPU
725         // So, globalItemSize has to be a multiple of localItemSize. 1024/64 = 16
726       ret = clEnqueueNDRangeKernel(partition1.commandQueue, partition1.kernel, 1, NULL, \
727                 &partition1.globalItemSize, &partition1.localItemSize, 0, NULL, NULL);
728         //ret = clEnqueueNDRangeKernel(partition1.commandQueue, partition1.kernel, 1, NULL, \
729                 &dataSIZE, &segmentSIZE, 0, NULL, NULL);
730       std::cout<<" End of execution!\n";
```

(Cont. Codigo fuente B.18) Archivo de Encabezado en OpenCL C/C++ con las declaraciones y definiciones de las Clases con los Métodos de Propósito General y Esqueletos Algorítmicos del API de SkeletonCoRe. Fuente: Elaborado por el autor.

```
731       // Copia de los datos de salida al Host (Copying output data back to the Host).
732       std::cout<<"    04.1  Copiando resultados de salida hacia el Host o Anfitrión!\n";
733       // Read from device back to host.
734       ret = clEnqueueReadBuffer(partition1.commandQueue, partition1.cMemObj, CL_TRUE, 0, \
735                 partition1.memVector_SIZE * sizeof(float), partition1.datvector_C, 0, NULL, NULL);
736         //std::cout<<"    04.2  ...fin de la ejecución!\n";
737     }
738     /*** Skeleton for Pipeline Parallel Processing ***/
739     void Skeletoncore_api::PipeSkeleton(SkelcorePartition &partition1, SkelcorePartition &partition2) {
740       // CPU Partition
741       ret = clEnqueueNDRangeKernel(partition1.commandQueue, partition1.kernel, 1, NULL, \
742                 &partition1.globalItemSize, &partition1.localItemSize, 0, NULL, NULL);
743         ret = clEnqueueReadBuffer(partition1.commandQueue, partition1.cMemObj, CL_TRUE, 0, \
744                 partition1.memVector_SIZE * sizeof(float), partition1.datvector_C, 0, NULL, NULL);
745         // GPU Partition
746         ret = clEnqueueNDRangeKernel(partition2.commandQueue, partition2.kernel, 1, NULL, \
747                 &partition2.globalItemSize, &partition2.localItemSize, 0, NULL, NULL);
748         ret = clEnqueueReadBuffer(partition2.commandQueue, partition2.cMemObj, CL_TRUE, 0, \
749                 partition2.memVector_SIZE * sizeof(float), partition2.datvector_C, 0, NULL, NULL);
750     }
751     /*** Skeleton for Pipeline Parallel Processing ***/
752     void Skeletoncore_api::PipeSkeleton(SkelcorePartition &partition1, SkelcorePartition &partition2,
753         SkelcorePartition &partition3) {
754       // CPU Partition
755       ret = clEnqueueNDRangeKernel(partition1.commandQueue, partition1.kernel, 1, NULL, \
756                 &partition1.globalItemSize, &partition1.localItemSize, 0, NULL, NULL);
757         ret = clEnqueueReadBuffer(partition1.commandQueue, partition1.cMemObj, CL_TRUE, 0, \
758                 partition1.memVector_SIZE * sizeof(float), partition1.datvector_C, 0, NULL, NULL);
759         // GPU Partition
760         ret = clEnqueueNDRangeKernel(partition2.commandQueue, partition2.kernel, 1, NULL, \
761                 &partition2.globalItemSize, &partition2.localItemSize, 0, NULL, NULL);
762         ret = clEnqueueReadBuffer(partition2.commandQueue, partition2.cMemObj, CL_TRUE, 0, \
763                 partition2.memVector_SIZE * sizeof(float), partition2.datvector_C, 0, NULL, NULL);
764     // FPGA Partition
765         ret = clEnqueueNDRangeKernel(partition3.commandQueue, partition3.kernel, 1, NULL, \
766                 &partition3.globalItemSize, &partition3.localItemSize, 0, NULL, NULL);
767         ret = clEnqueueReadBuffer(partition3.commandQueue, partition3.cMemObj, CL_TRUE, 0, \
768                 partition3.memVector_SIZE * sizeof(float), partition3.datvector_C, 0, NULL, NULL);
769     }
770     /*****************************************************************************/
771     //******* FIN DE DEFINICIÓN DE METODOS DE LA CLASE SKELETONCORE_API *******
772 }
773 //******* DECLARACION DEL OBJETO Y ESPACIO DE NOMBRES DE SKELETONCORE_API *******
774 using namespace skeletoncore_api;
775 Skeletoncore_api skelcore;        // Open and Calls skeletoncore API
776
777 //########################### FIN DEL API SKELETONCORE ############################
```

(Cont. Codigo fuente B.18) Archivo de Encabezado en OpenCL C/C++ con las declaraciones y definiciones de las Clases con los Métodos de Propósito General y Esqueletos Algorítmicos del API de SkeletonCoRe. Fuente: Elaborado por el autor.

C.3 Código fuente en OpenCL C/C++ de los Kernels usados en las Aplicaciones Paralelas de Prueba.

```
//#############################################################################################
//# SOBEL EDGE DETECTION KERNEL en OpenCL/C++ que contiene el Código de Cómputo Intensivo
//#############################################################################################
const sampler_t Sampler = CLK_NORMALIZED_COORDS_FALSE |
                CLK_ADDRESS_CLAMP |
                CLK_FILTER_NEAREST;
/*
// These are not actually needed, but included for reference
__constant int HorizontalFilter[9] = {
  1, 0, -1,
  2, 0, -2,
  1, 0, -1
};
__constant int VerticalFilter[9] = {
  1, 2, 1,
  0, 0, 0,
  -1, -2, -1
};
*/
// Runs the Sobel Filter on an image, this expects the image to be read in using unsigned integers to
// represent the RGB channels and should be between 0 and 255. This means the format CL_UNSIGNED_INT8
// should be used when creating the image2d_t source and output
__kernel void SobelFilter(
  __read_only image2d_t sourceImage,
  __write_only image2d_t outputImage,
  int width,
  int height
  )
{
  // This is the currently focused pixel and is the output pixel
  // location
  int2 ImageCoordinate = (int2)(get_global_id(0), get_global_id(1));

  // Make sure we are within the image bounds
  if (ImageCoordinate.x < width && ImageCoordinate.y < height) {
    int x = ImageCoordinate.x;
    int y = ImageCoordinate.y;

    // Read the 8 pixels around the currently focused pixel
    uint4 Pixel00 = read_imageui(sourceImage, Sampler, (int2)(x - 1, y - 1));
    uint4 Pixel01 = read_imageui(sourceImage, Sampler, (int2)(x, y - 1));
    uint4 Pixel02 = read_imageui(sourceImage, Sampler, (int2)(x + 1, y - 1));

    uint4 Pixel10 = read_imageui(sourceImage, Sampler, (int2)(x - 1, y));
    uint4 Pixel12 = read_imageui(sourceImage, Sampler, (int2)(x + 1, y));

    uint4 Pixel20 = read_imageui(sourceImage, Sampler, (int2)(x - 1, y + 1));
    uint4 Pixel21 = read_imageui(sourceImage, Sampler, (int2)(x, y + 1));
    uint4 Pixel22 = read_imageui(sourceImage, Sampler, (int2)(x + 1, y + 1));

    // This is equivalent to looping through the 9 pixels under this convolution and applying
    // the appropriate filter, here we've already applied the filter coefficients since they are static
    uint4 Gx = Pixel00 + (2 * Pixel10) + Pixel20 -
        Pixel02 - (2 * Pixel12) - Pixel22;

    uint4 Gy = Pixel00 + (2 * Pixel01) + Pixel02 -
        Pixel20 - (2 * Pixel21) - Pixel22;

    // Holds the output RGB values
    uint4 OutColor = (uint4)(0, 0, 0, 1);
```

Código fuente B.19: Código en OpenCL/C++ del Kernel para el operador Sobel's Edge Detection.
Fuente: Elaborado por el autor.

```
61
62      // Compute the gradient magnitude
63      OutColor.x = sqrt((float)(Gx.x * Gx.x + Gy.x * Gy.x)); // R
64      OutColor.y = sqrt((float)(Gx.y * Gx.y + Gy.y * Gy.y)); // G
65      OutColor.z = sqrt((float)(Gx.z * Gx.z + Gy.z * Gy.z)); // B
66
67      // Adjust all of the RGB values to not be more than 255
68      if (OutColor.x > 255) {
69        OutColor.x = 255;
70      }
71      if (OutColor.y > 255) {
72        OutColor.y = 255;
73      }
74      if (OutColor.z > 255) {
75        OutColor.z = 255;
76      }
77      // Convert to grayscale using luminosity method
78      uint Gray = (OutColor.x * 0.2126) + (OutColor.y * 0.7152) + (OutColor.z * 0.0722);
79
80      // Write the RGB value to the output image
81      write_imageui(outputImage, ImageCoordinate, (uint4)(Gray, Gray, Gray, 0));
82    }
83  }
84  /*
85  // Alternative way to generate Gx and Gy
86  int2 StartImageCoordinate = (int2)(ImageCoordinate.x - 1, ImageCoordinate.y - 1);
87  int2 EndImageCoordinate = (int2)(ImageCoordinate.x + 1, ImageCoordinate.y + 1);
88  uint4 Gx = (uint4)(0, 0, 0, 0);
89  uint4 Gy = (uint4)(0, 0, 0, 0);
90  int Index = 0;
91  for (int y = StartImageCoordinate.y; y < EndImageCoordinate.y; y++) {
92    for (int x = StartImageCoordinate.x; x < EndImageCoordinate.x; x++) {
93      uint4 Pixel = read_imageui(sourceImage, Sampler, (int2)(x, y));
94      Gx += Pixel * HorizontalFilter[Index];
95      Gy += Pixel * VerticalFilter[Index];
96      Index++;
97    }
98  }
99  */
```

(Cont. Codigo fuente B.19) Código en OpenCL/C++ del Kernel para el operador Sobel's Edge Detection. Fuente: Elaborado por el autor.

```
1  //##############################################################################
2  //# THRESHOLD FILTER KERNEL en OpenCL/C++ que contiene el Código de Cómputo Intensivo
3  //##############################################################################
4  __kernel void ThresholdFilter (__global const float2     *in,
5                                 __global const float      *aux,
6                                 __global       float2     *out,
7                                float value)
8  {
9    int gid = get_global_id(0);
10   float2 in_v  = in [gid];
11   float  aux_v = (aux)? aux[gid] : value;
12   float2 out_v;
13   out_v.x = (in_v.x > aux_v)? 1.0f : 0.0f;
14   out_v.y = in_v.y;
15   out[gid]  =  out_v;
16 }
```

Código fuente B.20: Código en OpenCL/C++ del Kernel para el operador Threshold. //Fuente: Elaborado por el autor.

```
//#############################################################################################
//# INTERPOLATION FILTER KERNEL en OpenCL/C++ que contiene el Código de Cómputo Intensivo
//#############################################################################################
__kernel void InterpolationFilter(
  __read_only image2d_t SourceImage,
  __write_only image2d_t DestinationImage,
  int width,
  int height,
  int scalingFactor)
{
    int row = get_global_id(0);
    int col = get_global_id(1);

    const int scaledWidth = width * scalingFactor;
    const int scaledHeight = height * scalingFactor;

    // Declaring sampler
    const sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE | CLK_FILTER_LINEAR | CLK_ADDRESS_CLAMP;

    float4 pixelValue = read_imagef(SourceImage, sampler, (int2)(col, row));

    for(int i = 0; i < scalingFactor; i++)
    {
        for(int j = 0; j < scalingFactor; j++)
        {
            write_imagef(DestinationImage, (int2)(?, ?), pixelValue);
        }
    }
}
```

Código fuente B.21: Código en OpenCL/C++ del Kernel para el operador Interpolation. //Fuente: Elaborado por el autor.