UNIVERSIDAD TÉCNICA DEL NORTE

Facultad de Ingeniería en Ciencias Aplicadas

Carrera de Software

OPTIMIZACIÓN DEL PROCESAMIENTO EN PARALELO UTILIZANDO PROGRAMACIÓN HETEROGÉNEA PARA MEJORAR EL RENDIMIENTO DE ALGORITMOS DE ALTO COSTE COMPUTACIONAL

Trabajo de grado previo a la obtención del título de Ingeniero de Software

Autor:

Riky Katari Tituaña Fernández

Director:

Ing. Cosme MacArthur Ortega Bustamante, MSc.

Ibarra – Ecuador

2024

UNIVERSIDAD TÉCNICA DEL NORTE

BIBLIOTECA UNIVERSITARIA

AUTORIZACIÓN DE USO Y PUBLICACIÓN

A FAVOR DE LA UNIVERSIDAD TÉCNICA DEL NORTE

1. IDENTIFICACIÓN DE LA OBRA

En cumplimiento del Art. 144 de la Ley de Educación Superior, hago la entrega del presente trabajo a la Universidad Técnica del Norte para que sea publicado en el Repositorio Digital Institucional, para lo cual pongo a disposición la siguiente información:

DATOS DE CONTACTO			
CÉDULA DE IDENTIDAD:	1003454681		
APELLIDOS Y NOMBRES:	Tituaña Fernández Riky Katari		
DIRECCIÓN:	Pukará de San Roque, Imbabura		
EMAIL:	rktituanaf@utn.edu.ec		
TELÉFONO FIJO:	N/A	**TELÉFONO MÓVIL:**	0982418926

DATOS DE LA OBRA	
TÍTULO:	Optimización del procesamiento en paralelo utilizando programación heterogénea para mejorar el rendimiento de algoritmos de alto coste computacional.
AUTOR (ES):	Tituaña Fernández Riky Katari
FECHA DE APROBACIÓN:	30/07/2024
PROGRAMA:	[X] **PREGRADO** [ ] **POSGRADO**
TITULO POR EL QUE OPTA:	INGENIERO EN SOFTWARE
DIRECTOR:	Ing. MacArthur Ortega-Bustamante, MSc.
ASESOR 1:	PhD. Marco Remigio Pusdá Chulde

2. CONSTANCIAS

El autor (es) manifiesta (n) que la obra objeto de la presente autorización es original y se la desarrolló, sin violar derechos de autor de terceros, por lo tanto la obra es original y que es (son) el (los) titular (es) de los derechos patrimoniales, por lo que asume (n) la responsabilidad sobre el contenido de la misma y saldrá (n) en defensa de la Universidad en caso de reclamación por parte de terceros.

Ibarra, a los 30 días del mes de julio de 2024

EL AUTOR:

ESTUDIANTE
Riky Katari Tituaña Fernández
C.I.: 1003454681

CERTIFICACIÓN DIRECTOR

Ibarra, 30 de julio del 2024

CERTIFICACIÓN DIRECTOR DEL TRABAJO DE TITULACIÓN

Por medio del presente yo Ing. MacArthur Ortega-Bustamante, MSc., certifico que el Sr. Riky Katari Tituaña Fernández portador de la cedula de ciudadanía número 1003454681, ha trabajado en el desarrollo del proyecto de grado “Optimización del procesamiento en paralelo utilizando programación heterogénea para mejorar el rendimiento de algoritmos de alto coste computacional.”, previo a la obtención del Título de Ingeniero en Software realizado con interés profesional y responsabilidad que certifico con honor de verdad.

Es todo en cuanto puedo certificar a la verdad

Atentamente

1001580396 COSME MACARTHUR ORTEGA BUSTAMANTE

Firmado digitalmente por 1001580396 COSME MACARTHUR ORTEGA BUSTAMANTE
Fecha: 2024.07.30 15:14:59 -05'00'

Ing. MacArthur Ortega-Bustamante, MSc.

DIRECTOR DE TRABAJO DE GRADO

Dedicatoria

A mis padres, Beatriz Fernández y Alberto Tituaña por su amor incondicional y estar presentes en cada paso de mi vida, por darme su apoyo y confianza, por guiarme en el camino de la vida con sus consejos, por enseñarme la importancia de la perseverancia y que lo más importante es la familia. A mi tía, Jesusa Fernández que me ayudó gran parte de mi vida universitaria. A mis hermanos, por brindarme su apoyo en las distintas etapas de mi vida y extenderme una mano cuando los necesitaba. Aunque algunas veces tengamos desacuerdos, como cualquier persona, los quiero mucho y siempre los querré.

A mi familia, que forman parte de mi vida y que me brindaron palabras de aliento y me empujaron a seguir creciendo como persona.

A mis amigos, compañeros y docentes, que fueron parte importante en mi etapa universitaria, con quienes compartí alegrías, experiencias y enseñanzas.

Riky Katari Tituaña Fernández

Agradecimientos

Agradezco a mis padres, quienes han sido mis guías, compartiendo conmigo su conocimiento y sabiduría, y mostrándome siempre el camino correcto. Gracias a su apoyo, he podido cumplir mis metas y me siento afortunado de tener unos padres tan maravillosos. A mis queridos hermanos, por compartir tantas experiencias juntos y por estar siempre ahí para ayudarme cuando los necesito. A mi tía, por su bondad y amabilidad, por ayudarme a mí y a mi familia sin esperar nada a cambio. Le estaré eternamente agradecido.

Le agradezco a mi familia, por ser parte importante de mi vida, preocuparse por mí. A mis abuelos, tíos y primos les agradezco por compartir buenos momentos en familia, buenas experiencias, por ser siempre unidos en los buenos y malos momentos.

Al MSc. Cosme Ortega quien fue mi maestro y tutor, le agradezco por su tiempo, por haberme apoyado con su conocimiento y gestión en el transcurso del desarrollo de mi trabajo de grado.

Agradezco a la Universidad Técnica del Norte, en especial a la Facultad de Ingeniería en Ciencias Aplicadas (FICA), por brindarme la oportunidad de formarme como profesional y mejorar como persona. A lo largo de mi vida estudiantil he ido adquiriendo conocimiento, habilidades y experiencia que me servirán por el resto de mi vida, por ello estoy profundamente agradecido con la comunidad universitaria de la gloriosa UTN.

Riky Katari Tituaña Fernández

Tabla de contenido

Índice de Figuras

Índice de Tablas

Resumen

El presente caso de estudio tiene como objetivo el análisis de la programación heterogénea para optimizar algoritmos de alto coste computacional. El documento está conformado en tres capítulos en los cuales se detalla todo el proceso para realizar el Trabajo de Grado: “OPTIMIZACIÓN DEL PROCESAMIENTO EN PARALELO UTILIZANDO PROGRAMACIÓN HETEROGÉNEA PARA MEJORAR EL RENDIMIENTO DE ALGORITMOS DE ALTO COSTE COMPUTACIONAL.”

En la Introducción del presente documento se abordan los siguientes aspectos, antecedentes, prospectiva, planteamiento del problema, situación actual, objetivo general y específico, alcance y justificación. Se destaca la creciente necesidad de mejorar el rendimiento de algoritmos de alta complejidad en diversos campos.

En el primer capítulo se encuentra documentado el marco teórico en donde se realizó el estudio de diferentes arquitecturas de procesadores como CPU, GPU y FPGA, además del procesamiento en paralelo. Se seleccionó la plataforma de Intel DevCloud y el lenguaje de programación heterogénea DPC++ en donde se implementaron las pruebas de rendimiento de los algoritmos de alto coste computacional.

El capítulo dos se centra en el desarrollo e implementación de algoritmos de alto coste computacional. Los algoritmos seleccionados son: algoritmo de Matriz de Distancia Euclidiana y el algoritmo de All Pairs Shortest Paths de Floyd-Warshall. Se realizaron pruebas de rendimiento a los algoritmos en diferentes arquitecturas utilizando programación heterogénea para optimizar el procesamiento en paralelo. Los resultados obtenidos demuestran una mejora significativa en la velocidad de procesamiento, la escalabilidad y la eficiencia en la utilización de recursos de hardware con respecto a la programación secuencial.

Finalmente, en el último capítulo se valida y analiza los resultados obtenidos en base al promedio de tiempo de procesamiento en segundos y la aceleración, donde la implementación de los algoritmos en paralelo muestras una evidente mejora en rendimiento frente a los algoritmos en forma secuencial.

Palabras claves: optimización, programación heterogénea, DPC++, procesamiento en paralelo, FPGA, rendimiento, alto coste computacional.

Abstract

The objective of this case study is the analysis of heterogeneous programming to optimize high computational cost algorithms. The document is conformed in three chapters in which the whole process to carry out the Degree Work is detailed: "OPTIMIZATION OF PARALLEL PROCESSING USING HETEROGENEOUS PROGRAMMING TO IMPROVE THE PERFORMANCE OF HIGH COMPUTATIONAL COST ALGORITHMS".

The Introduction of this document addresses the following aspects, background, prospective, problem statement, current situation, general and specific objective, scope and justification. It highlights the growing need to improve the performance of highly complex algorithms in various fields.

The first chapter documents the theoretical framework where the study of different processor architectures such as CPU, GPU and FPGA, as well as parallel processing, was carried out. The Intel DevCloud platform and the heterogeneous programming language DPC++ were selected where the performance tests of high computational cost algorithms will be implemented.

Chapter two focuses on the development and implementation of high computational cost algorithms. The selected algorithms are: Euclidean Distance Matrix algorithm and the Floyd-Warshall All Pairs Shortest Paths algorithm. Performance tests were performed on the algorithms on different architectures using heterogeneous programming to optimize parallel processing. The results obtained show a significant improvement in processing speed, scalability and efficiency in hardware resource utilization with respect to sequential programming.

Finally, the last chapter validates and analyzes the results obtained based on the average processing time in seconds and acceleration, where the implementation of parallel algorithms shows a clear improvement in performance over sequential algorithms.

Keywords: optimization, heterogeneous programming, DPC++, parallel processing, FPGA, performance, high computational cost.

INTRODUCCIÓN

Antecedentes

La computación de alto rendimiento ha crecido de manera significativa en las últimas décadas, debido a la necesidad de resolver problemas complejos y manejar grandes volúmenes de datos en áreas como la bioinformática, la simulación de fenómenos físicos y el análisis de grandes volúmenes de datos (big data). Tradicionalmente, los algoritmos de alto coste computacional se han desarrollado en arquitecturas de CPU, utilizando técnicas de paralelización como el procesamiento multihilo (Hyper-Threading) y computación distribuida. Sin embargo, la evolución de las arquitecturas de hardware ha abierto nuevas oportunidades para mejorar el rendimiento a través de la programación heterogénea, que combina diferentes tipos de procesadores como CPU, GPU y FPGA, para optimizar la ejecución de tareas.

Situación actual

Actualmente, la programación heterogénea es una tendencia frecuente en el desarrollo de aplicaciones de alto rendimiento. Las herramientas y lenguajes como DPC++ (Data Parallel C++), que forma parte del ecosistema oneAPI de Intel, permiten a los desarrolladores aprovechar las capacidades de varios tipos de hardware en una única aplicación. La programación heterogénea agiliza la distribución de tareas de cálculo intensivo a dispositivos especializados, mejorando así la velocidad de procesamiento y la eficiencia en la optimización de dichos recursos. A pesar de los avances, la aplicación eficaz de algoritmos de alto coste computacional en entornos heterogéneos sigue planteando importantes desafíos, como la necesidad de optimizaciones específicas y de una correcta gestión de la memoria y sincronización entre dispositivos.

Los sistemas informáticos modernos de las grandes empresas necesitan procesar grandes volúmenes de información, no pudiendo realizarlo en arquitecturas personales debido a que las aplicaciones que se ejecutan se sustentan en algoritmos de alto coste computacional, algoritmos que no son eficientes porque no utilizan técnicas avanzadas y especializadas de programación que permitan optimizar los tiempos de ejecución y los recursos del hardware. Por otra parte, existen algoritmos de alta complejidad que deben realizar cálculos muy complejos para obtener un resultado adecuado, la ejecución de estos algoritmos puede tomar mucho tiempo y los lenguajes de programación convencionales y el hardware doméstico no están diseñados para ejecutar este tipo de algoritmos, estas computadoras no cuentan con aceleradores heterogéneos como una GPU o FPGA (Ramírez Patiño & Guerrero Hernández, 2017).

Prospectiva

La evolución hacia una arquitectura de hardware heterogénea promete una nueva revolución en el campo de la computación de alto rendimiento. Con el desarrollo continuo de herramientas y lenguajes de programación que facilitan la implementación y optimización de algoritmos en entornos heterogéneos, se espera que más aplicaciones complejas se beneficien de estas nuevas tecnologías. Además, la capacidad de escalabilidad eficaz tanto vertical como horizontal permitirá a las organizaciones manejar cada vez problemas más grandes y complejos. La investigación de nuevas técnicas y metodologías para optimizar el rendimiento y la productividad en entornos heterogéneos continuará siendo un área clave de desarrollo.

La presente propuesta de investigación plantea optimizar la ejecución de algoritmos de alta complejidad empleando programación heterogénea, debido a que con el pasar del tiempo, los programas computacionales se vuelven cada vez más robustos, por lo cual, aumenta la necesidad de nuevas tecnologías que permitan optimizar el tiempo de ejecución y aprovechar mejor los recursos de hardware.

La finalidad de la propuesta es apoyar a la comunidad de investigación científica sobre temas de optimización de algoritmo y aprovechamiento del hardware computacional, se creó un proyecto de investigación sobre la aplicación de algoritmos de alto rendimiento en sistemas heterogéneos.

Planteamiento del problema

Cuando el cálculo o el procesamiento de los datos es de un alto coste computacional, el sistema informático podría tardar demasiado tiempo o en el peor de los casos colapsar. Ejemplos claros son los programas de predicción del clima o algoritmos de matemática computacional, que recibe una gran cantidad de información y una cantidad exagerada de cálculos, que posteriormente debe ser procesada (Requene, 2022).

Se ha detectado un bajo nivel de aprovechamiento y desconocimiento acerca del Computación de Alto Rendimiento (HPC) y de la programación heterogénea. Los programas y algoritmos desarrollados no están optimizados para utilizar de manera eficiente los sistemas de computación de alto rendimiento. En la Figura 1 se detalla las causas y efectos que se han detectado en el planteamiento del problema.

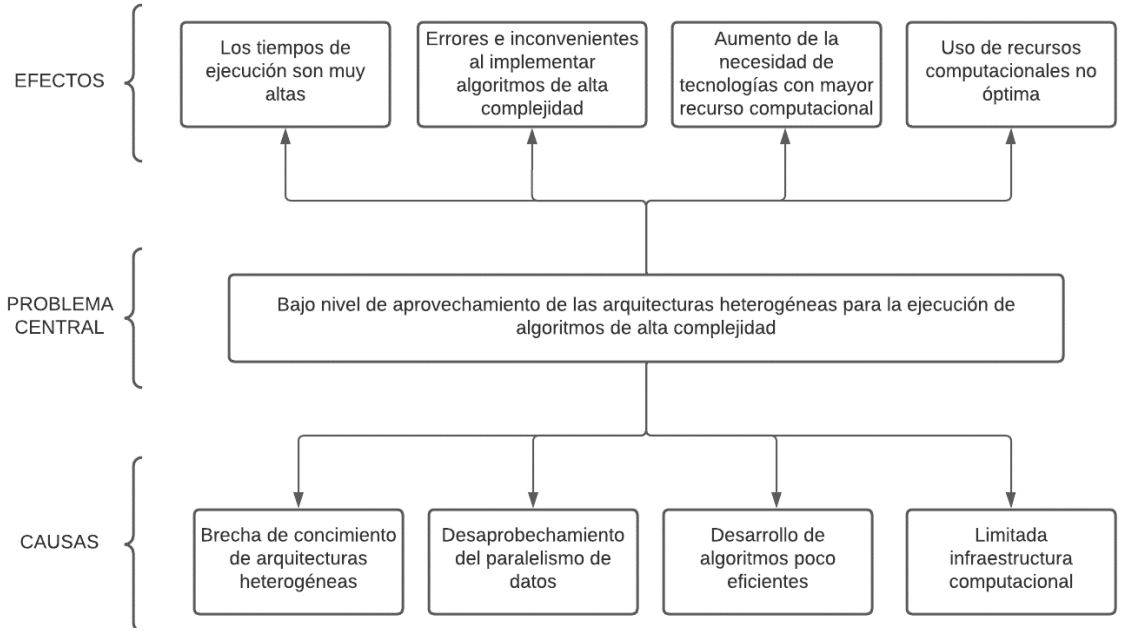


Figura 1. Árbol de problemas

Fuente: Elaboración propia

Objetivos

Objetivo general

Optimizar el procesamiento en paralelo utilizando programación heterogénea para mejorar el rendimiento de algoritmos de alto coste computacional.

Objetivo específico

- Documentar un marco teórico sobre las arquitecturas y programación heterogénea y las herramientas para la ejecución de algoritmos de programación paralela.
- Medir el rendimiento al implementar algoritmos de gran complejidad utilizando lenguajes de programación para arquitecturas heterogéneas.
- Validar los resultados obtenidos utilizando métricas y métodos de evaluación definidas en la investigación.

Alcance

El proyecto tiene como finalidad medir los costes computacionales al implementar algoritmos de alta complejidad en arquitecturas de HPC. Para ello se podría utilizar lenguajes de programación heterogénea tales como DPC++ (Reinders et al., 2021). Se implementarán como mínimo 2 algoritmos de alta complejidad, las cuales se definirán en la etapa de investigación.

Para el desarrollo del proyecto se utilizó Intel Developer Cloud la cual ofrece acceso gratuito a una amplia gama de arquitecturas Intel para ayudar a obtener una experiencia

práctica instantánea con el software Intel y ejecutar sus cargas de trabajo de renderizado, IA, computación de alto rendimiento (HPC) y edge. Con marcos, herramientas y bibliotecas optimizados de Intel preinstalados, tiene todo lo que necesita para acelerar su aprendizaje y la creación de prototipos de proyectos (Intel Corporation, 2021).

Mediante el estudio de técnicas y lenguajes de programación heterogénea se implementaron algoritmos de alto coste computacional en arquitecturas de HPC, las cuales cuentan con hardware moderno que no incluye solo la CPU, sino también distintos aceleradores heterogéneos, como la GPU y FPGA (Soto, 2016). Se definieron métricas de evaluación para medir la optimización de los algoritmos en base a las recomendaciones sugeridas en la investigación de programación sobre sistemas de computación de alto rendimiento (Ramírez Patiño & Guerrero Hernández, 2017).

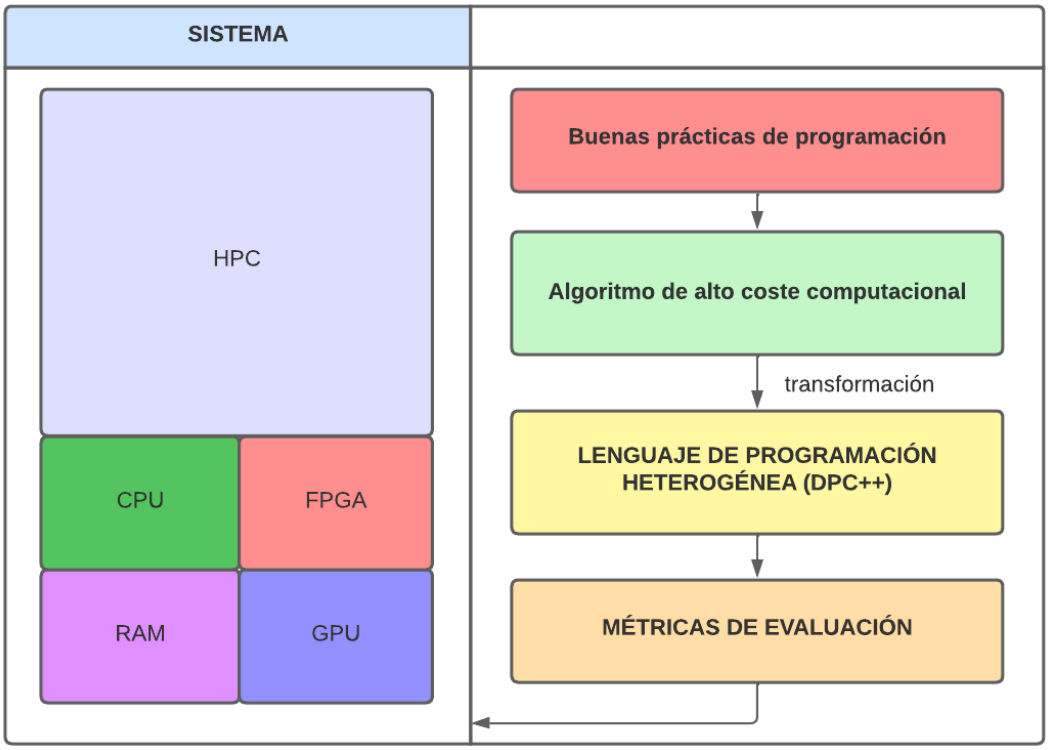


Figura 2. Diagrama de proceso.

Fuente: Elaboración propia

Método de investigación

Para el desarrollo de este proyecto se utilizó el método científico que sirvió como guía para responder a la pregunta de investigación que se plantearon en el desarrollo del proyecto y la creación de hipótesis porque usa un conjunto de técnicas y procedimientos para la obtención de resultados fiables dando soluciones a los problemas de investigación.

Para el cumplimiento del primer objetivo, el marco teórico se desarrolló con un fin investigo/analítico de tipo empírico, mediante numerosas fuentes bibliográficas sobre la

programación heterogénea, algoritmos de alto coste computacional, arquitecturas de HPC, procesamiento en paralelo y aceleradores heterogéneos (Guamán Rivera, 2017). Todas las referencias utilizadas para esta fase investigativa son provenientes de fuentes científicas, veraces y confiables.

El segundo objetivo se lo llevó a cabo utilizando los conocimientos abstraídos en la primera fase del proyecto, se implementaron algoritmos de alta complejidad computacional en las herramientas que nos ofrece Intel® Developer Cloud y se desarrollará las siguientes tareas:

- Preparación del entorno de ejecución en una arquitectura de HPC y con un lenguaje de programación heterogénea.
- Aplicación de buenas prácticas de programación.
- Ejecución de algoritmos de alto coste computacional en el entorno preparado anteriormente.

Para cumplir el objetivo 3, se desarrolló una investigación cuantitativa, la cual midió los resultados obtenidos al momento de ejecutar los algoritmos con las métricas y métodos de evaluación definidas en el desarrollo de la investigación.

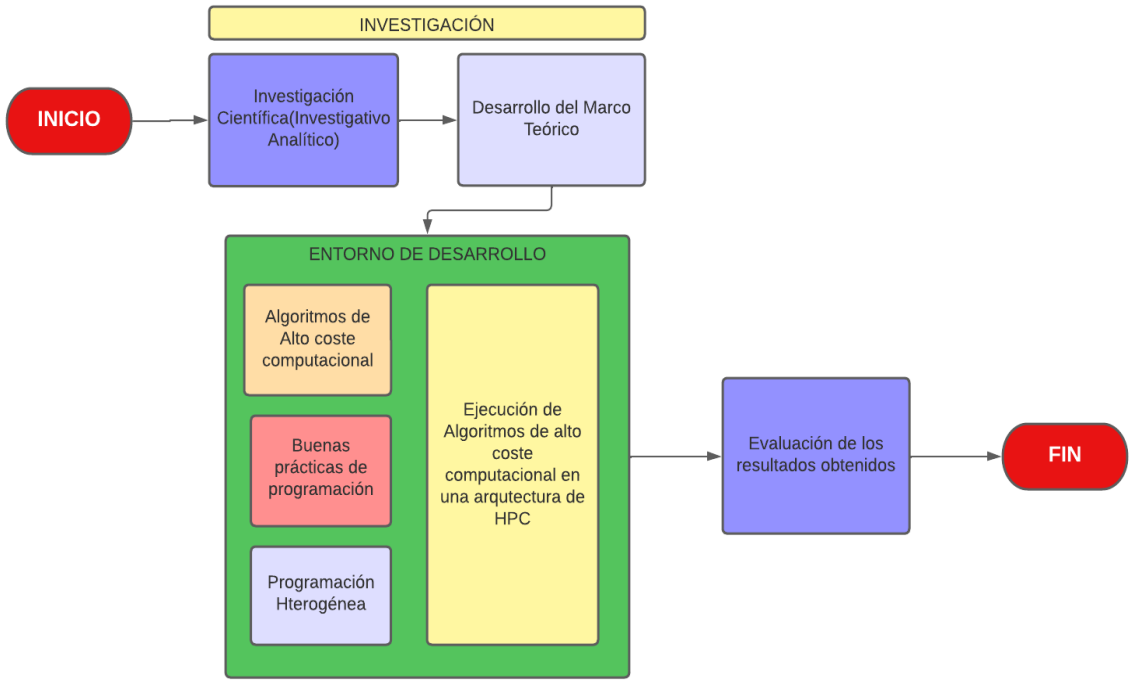


Figura 3. Metodología del Proyecto.

Fuente: Elaboración propia

Justificación

Justificación Política

El proyecto se enfoca en el objetivo 9: Industria, innovación e Infraestructura de los Objetivos de Desarrollo Sostenible:

9.4 De aquí a 2030, modernizar la infraestructura y reconvertir las industrias para que sean sostenibles, utilizando los recursos con mayor eficacia y promoviendo la adopción de tecnologías y procesos industriales limpios y ambientalmente racionales, y logrando que todos los países tomen medidas de acuerdo con sus capacidades respectivas (Organización de las Naciones Unidas, 2018).

9.5 Aumentar la investigación científica y mejorar la capacidad tecnológica de los sectores industriales de todos los países, en particular los países en desarrollo, entre otras cosas fomentando la innovación y aumentando considerablemente, de aquí a 2030, el número de personas que trabajan en investigación y desarrollo por millón de habitantes y los gastos de los sectores público y privado en investigación y desarrollo (Organización de las Naciones Unidas, 2018).

Justificación Tecnológica

Una de las áreas de interés que están en crecimiento es el cómputo de altas prestaciones, en el cual el rendimiento está relacionado con dos aspectos fundamentales: por un lado, las arquitecturas de soporte y por otro lado los algoritmos que hacen uso de estas (Naiouf et al., 2015).

La aparición de arquitecturas multi-core (como las GPU o los procesadores MIC), se ha sumado el uso de FPGAs debido a su potencia de cómputo y rendimiento energético (Ramírez Patiño & Guerrero Hernández, 2017).

Lógicamente, esto trae aparejado una revisión de los conceptos del diseño de algoritmos paralelos (incluyendo los lenguajes mismos de programación y el software de base), así como la evaluación de las soluciones que éstos implementan (Naiouf et al., 2015).

Justificación Teórica

La línea de investigación sobre implantación de algoritmos de alto coste computacional mediante programación heterogénea fomentará la adopción de nuevas tecnologías. De este modo, se abrirán nuevas opciones para los desarrolladores implicados en la investigación y optimización de algoritmos, lo que permitirá desarrollar soluciones más avanzadas y eficientes.

Contexto

Tabla 1

Contexto de investigación

Contexto local, nacional e internacional en base a tesis, trabajos o investigaciones realizadas:		
INVESTIGACION	**ENLACE**	**APORTE**
Contexto: Local Implementación de algoritmos sobre arquitectura multinúcleo para optimizar el alto coste computacional al procesar grandes volúmenes de datos.	http://repositorio.utn.edu.ec/handle/123456789/13091	La investigación se centra en la optimización de algoritmos pesados en computadoras de HPC, mientras que el presente proyecto se centra en la implementación de programación paralela en arquitecturas heterogéneas.
Contexto: Local Comparativa de los paradigmas de programación paralela: por manejo de threads, por paso de mensajes e híbrida.	http://repositorio.utn.edu.ec/handle/123456789/7729	Comparación de diferentes leguajes de programación paralela y el manejo de Threads.
Contexto: Nacional Análisis de rendimiento de un clúster HPC y, arquitecturas manycore y multicore.	http://dspace.ucuenca.edu.ec/bitstream/123456789/28554/1/Trabajo%20de%20Titulaci%C3%B3n.pdf	Esta investigación se centra en el análisis de clúster de HPC, el proyecto planteado utilizará una los aceleradores heterogéneos de las computadoras de HPC para poder ejecutar algoritmos de programación paralela.

Contexto: Nacional Arquitectura Clúster de Alto Rendimiento Utilizando Herramientas de Software Libre.	https://lajc.epn.edu.ec/index. php/LAJC/article/view/59	El trabajo encontrado se centra en la utilización de un HPC, la propuesta plateada se centra en la programación en estas arquitecturas.
Contexto: Internacional Programación paralela sobre arquitecturas heterogéneas.	https://repositorio.unal.edu.co/ handle/unal/57830	La investigación encontrada realiza pruebas en un entorno de pruebas diferente al del presente proyecto.
Contexto: Internacional Análisis del impacto de la arquitectura multi-núcleo en cómputo paralelo.	https://tesis.ipn.mx/bitstream/ handle/123456789/5918/1411 .pdf?sequence=1&isAllowed=y	La investigación encontrada se centra en los procesadores multi-core mientras el proyecto propuesto se centra en las arquitecturas heterogéneas.

Nota: Elaboración propia

CAPÍTULO 1

Marco Teórico

La Tabla 2 presenta de manera organizada el contenido del marco teórico, el cual define los conceptos fundamentales relacionados con la optimización del procesamiento en paralelo utilizando programación heterogénea para mejorar el rendimiento de algoritmos de alto coste computacional".

Tabla 2

Esquematización del marco teórico

Tema principal	Tema Secundario	Tema complementario
Arquitecturas heterogéneas	CPU	Hyper-Threading
	GPU	
	FPGA	FPGA en la actualidad
Estudio del procesamiento en paralelo	Paralelismo	
	Factor SpeedUp	
	Ley de Amdahl	
	Procesamiento en paralelo	
	Tipos de procesamiento en paralelo	Instrucción Única, Datos Únicos (SISD)
		Instrucción única,

		datos múltiples (SIMD) Instrucción múltiple, datos únicos (MISD) Instrucción múltiple, datos múltiples (MIMD) Programa único, datos múltiples (SPMD) Procesamiento masivo en paralelo (MPP)
Programación heterogénea	DPC++ Queues y Actions	
Algoritmos de alto coste computacional	Complejidad computacional Complejidad algorítmica	
	Notación Big O	Complejidad lineal - O(n) Complejidad exponencial - O(n^2), O(n^3), etc. Complejidad logarítmica O (log n) Complejidad constante - O (1)
Buenas prácticas de programación	Metas de la MPP Fases de la MPP	

Nota: Elaboración propia

1.1. Arquitecturas heterogéneas

Se puede entender como arquitectura heterogénea a un tipo de sistema que incluye diferentes tipos de unidades de procesamiento, como por ejemplo CPU multinúcleo, GPU de muchos núcleos de uso general y FPGAs programables, cada una de ellas están diseñadas para desempeñar una tarea específica (Gizopoulos et al., 2019).

El acelerador más común utilizado en arquitecturas heterogéneas es la GPU, sin embargo, este dispositivo requiere un consume energético elevado comparado con una FPGA. Por esta razón, existen computadoras de alto rendimiento los cuales tienen integrado una FPGA. Estos dispositivos al ser semiconductores que consumen poca energía y son eficientes (Valdez & Maldonado, 2021).

1.1.1. CPU

La CPU o Unidad Central de Procesamiento por sus siglas en inglés (Central Processing Unit) es la encargada de llevar a cabo tareas de procesamiento, principalmente la ejecución de programas. Sin embargo, las CPU actuales presentan funcionalidades avanzadas, como la inclusión de múltiples núcleos y capacidades de hiper-procesamiento (en inglés, "hyper-threading"). En algunas computadoras, incluso se emplean múltiples unidades de procesamiento central (Lewis & Hoffman, 2023).

En la Figura 4 se muestra como está conformado un procesador multinúcleo. También se muestra cómo es que un sistema operativo host asigna varias aplicaciones a un hardware con una arquitectura de 4 núcleos homogéneos (Requene, 2022).

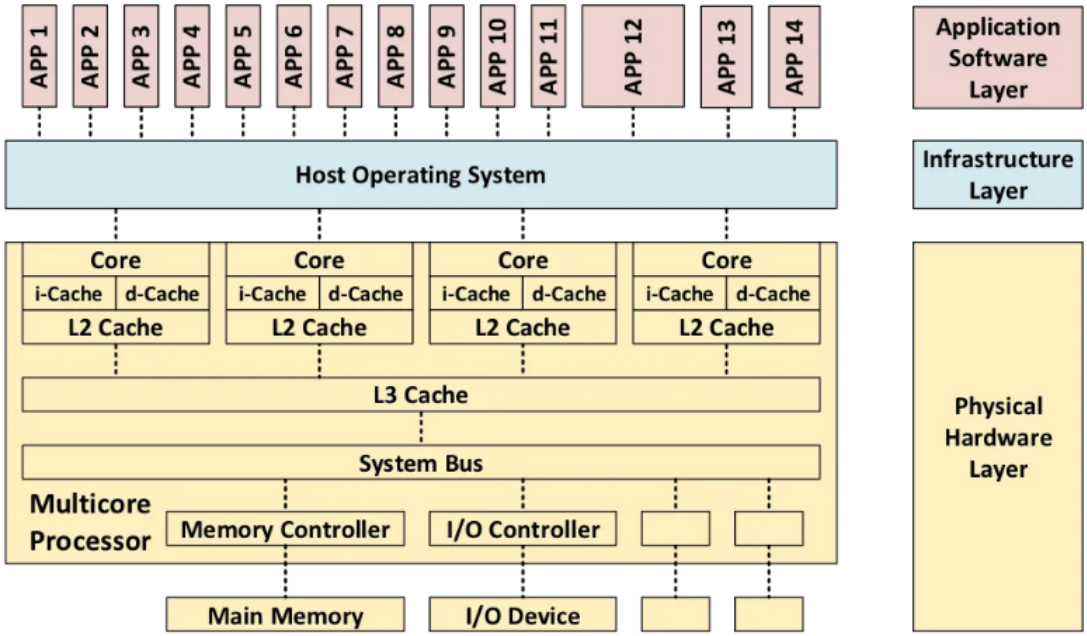


Figura 4. Arquitectura de una CPU multinúcleo

Fuente: (Firesmith, 2017)

Hyper-Threading

La tecnología Hyper-Threading representa un avance en hardware que posibilita la ejecución simultánea de múltiples subprocesos en un mismo núcleo, lo que conlleva a una mayor capacidad de realizar tareas de forma simultánea, es decir en paralelo (Intel Corporation, 2020).

Es importante destacar que la tecnología Hyper-Threading es un desarrollo exclusivo de Intel, y solo los procesadores de esta marca integran esta característica. Por otro lado, en el caso de las CPUs de AMD, se encuentra una tecnología análoga denominada "SMT" (Simultaneous Multi-Threading) que opera de forma semejante al Hyper-Threading de Intel (Lewis & Hoffman, 2023). Esta tecnología posibilita que un núcleo físico emule varios núcleos lógicos, mejorando así la eficacia en la ejecución de tareas.

El sistema operativo Windows cuenta con un aparatado en el administrador de tareas para monitorear el rendimiento de un sistema informático, contiene la información del procesador, de la GPU en caso de tener una, entre otras especificaciones de hardware como de muestra en la Figura 5.

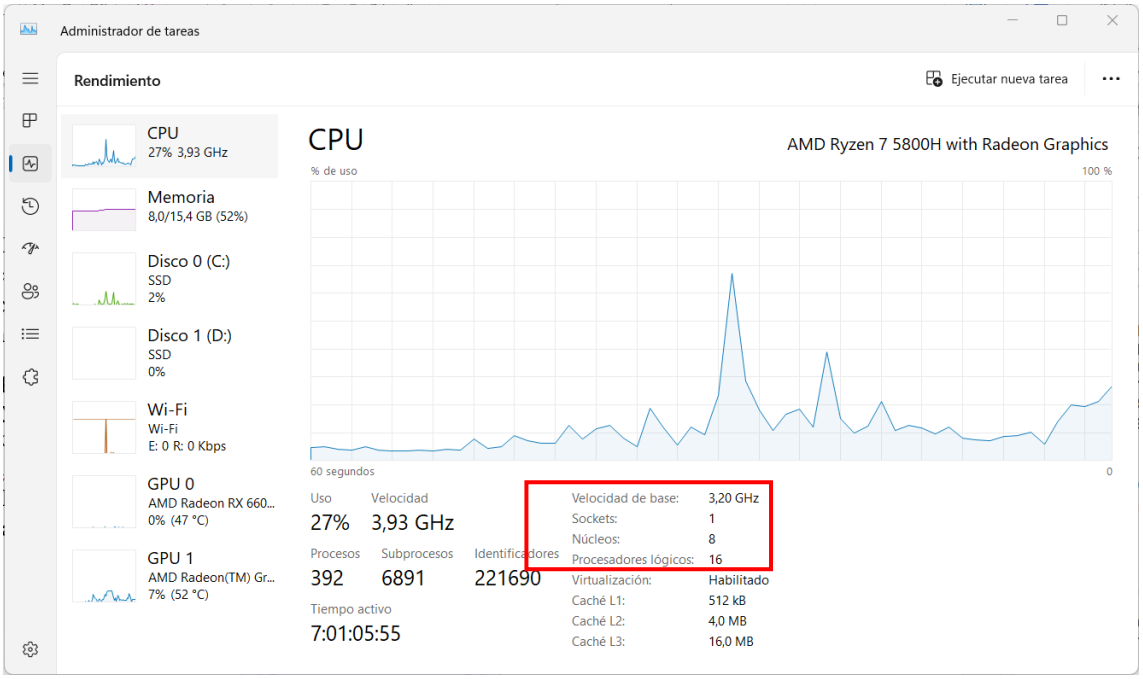


Figura 5. Características de una CPU desde el administrador de tareas de Windows

Fuente: Elaboración propia

La tecnología de Intel Hyper-Threading al estar activa, hace que el Sistema Operativo identifique dos contextos de ejecución o procesadores lógicos por cada núcleo físico. De esta manera cada núcleo físico puede manejar diferentes subprocesos de software ya que funciona como dos núcleos lógicos (Intel Corporation, 2020).

Gracias a la tecnología de CPU con Hyper-Threading (Figura 6), una computadora tiene la capacidad de manejar una mayor cantidad de datos en un periodo de tiempo más corto y realizar múltiples tareas en segundo plano sin interrupciones. En condiciones apropiadas, esta tecnología permite que los núcleos de la CPU ejecuten eficazmente dos actividades simultáneas (Intel Corporation, 2020).

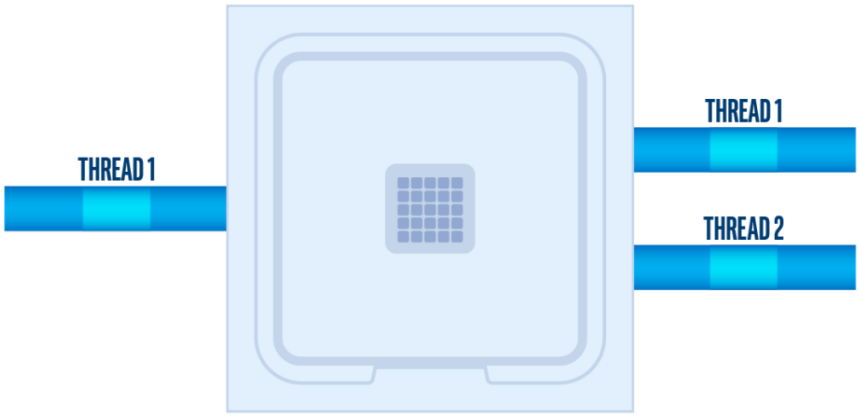


Figura 6. Hyper-Threading

Fuente: (Intel Corporation, 2020)

1.1.2. GPU

La GPU o Unidad de Procesamiento Gráfico por sus siglas en inglés (Graphics Processing Unit), comúnmente reconocida como un componente de hardware esencial para ejecutar aplicaciones que demandan un alto procesamiento gráfico, como el software de modelado 3D o la infraestructura de escritorio virtual (VDI). En el ámbito de consumo, la GPU se utiliza principalmente para potenciar el rendimiento gráfico de los videojuegos. Actualmente, las GPU de uso general (GPGPU) son la elección preferida en el hardware para acelerar las tareas de cómputo en los modernos entornos de alto rendimiento informático (HPC) (Hagoort, 2020).

Al explorar la arquitectura de una GPU de alto nivel (que depende en gran parte de la marca y el modelo), se entiende que la GPU se basa en utilizar los núcleos que tiene disponibles más que en el acceso a memoria caché de baja latencia (Hagoort, 2020).

Una GPU consta de múltiples clústeres de procesadores, cada uno con varios multiprocesadores de transmisión (SM). Cada SM tiene su propia caché de instrucciones de nivel 1 y comparte una caché de nivel 2 antes de acceder a la memoria global como se muestra en la Figura 7. A diferencia de las CPU, las GPU tienen menos y más pequeñas capas de memoria caché debido a su enfoque en el procesamiento de datos en lugar de la latencia de memoria. Su diseño está optimizado para cálculos de alto rendimiento en paralelo (Hagoort, 2020).

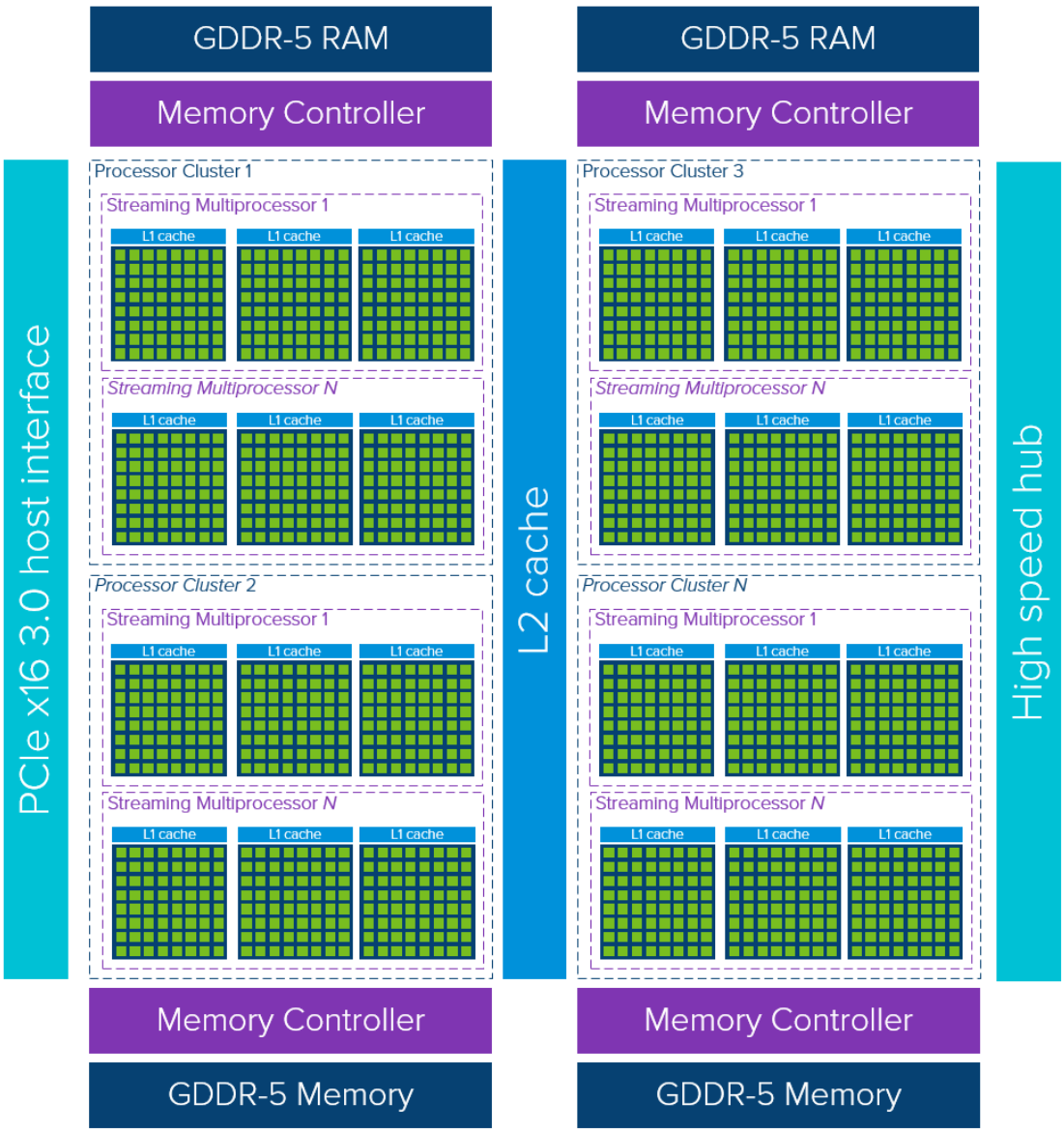


Figura 7. Arquitectura de una GPU

Fuente: (Hagoort, 2020)

1.1.3. FPGA

Las FPGAs (Field-Programmable Gate Array) o Matriz de Puertas Programable en Campo en español, son chips de silicio reprogramable, esto significa que, a diferencia de los procesadores convencionales de una computadora, la programación de un FPGA implica la reconfiguración del chip para establecer su funcionalidad en lugar de simplemente ejecutar una aplicación de software. La FPGA, inventada en 1985 por Ross Freeman, cofundador de Xilinx, se compone de una matriz de Bloques de Lógica Configurables (CLBs) que están interconectados, junto con bloques de entrada y salida (IOBs) (Valdez & Maldonado, 2021).

Estos dispositivos de hardware programable albergan celdas de lógica en su interior, las cuales pueden ser configuradas y conectadas según las necesidades a través de un lenguaje de descripción de hardware (HDL, Hardware Description Language).(Daroch & Antiñire, 2018).

En la Figura 8 se presentan los atributos de las FPGA correspondientes a 1988, tal como fueron divulgados por Xilinx. Esto permite visualizar cómo ha avanzado la tecnología de las FPGA con el tiempo. Los atributos incluyen la capacidad, que se refiere al número de celdas lógicas; la velocidad, que representa el rendimiento; el precio, calculado en función de cada celda lógica; y la eficiencia energética, también evaluada por celda lógica (Daroch & Antiñire, 2018).

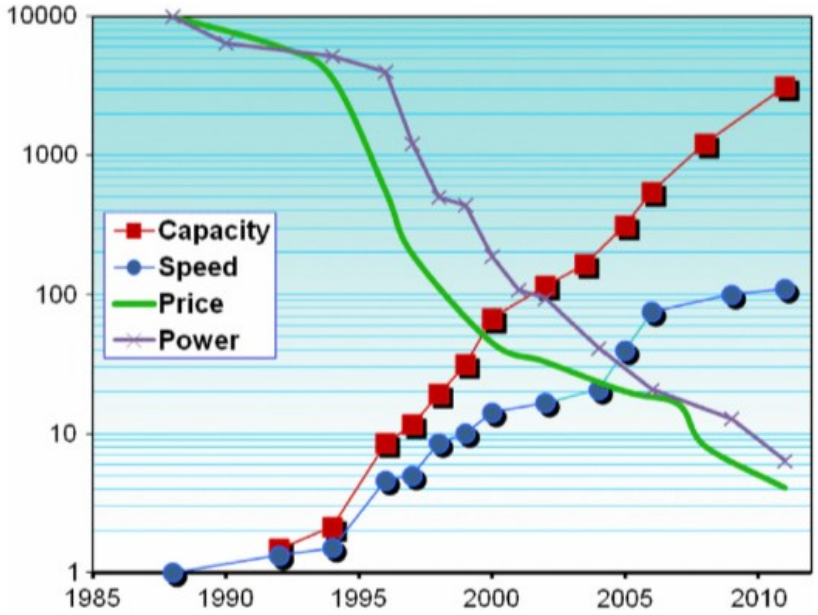


Figura 8. Atributos de FPGA de Xilinx relativos a 1988.

Fuente: (Trimberger, 2018)

La primera FPGA desarrollada fue el Xilinx XC2064, que presentaba las siguientes características: contaba con 64 bloques lógicos complejos (CLB) dispuestos en una matriz de 8x8, una velocidad de 18MHz, 58 pines de entrada/salida (I/O pins), 1200 puertas lógicas y era un circuito integrado de considerable tamaño. En aquel momento, el tamaño de la matriz y el costo por función eran aspectos cruciales. Aunque el XC2064 parece modesto en comparación con las FPGA actuales, su valor en ese entonces era significativo, llegando a costar cientos de dólares (Daroch & Antiñire, 2018).

Una FPGA es un dispositivo integrado que alberga componentes lógicos y conexiones programables entre ellos. Estos componentes lógicos pueden configurarse como compuertas lógicas básicas como por ejemplo AND, OR, XOR y NOT, o inclusive funciones de combinación más complejas, como podrían ser los decodificadores, además de permitir la implementación de funciones matemáticas sencillas. Además, las FPGA también incorporan elementos de memoria, que pueden variar desde flip-flops sencillos hasta bloques de memoria de mayor complejidad (Daroch & Antiñire, 2018).

FPGA en la actualidad

Para comprender la importancia que las FPGAs han adquirido en la actualidad, es crucial resaltar los eventos más significativos que han tenido un impacto en la industria de desarrollo de estos chips.

En la actualidad, la perspectiva de la supercomputación basada en FPGA se encuentra en un momento excepcionalmente prometedor debido a una serie de factores clave. Estos factores incluyen avances en la potencia de los chips FPGA, el desarrollo maduro de herramientas de programación como OpenCL, la adquisición de Altera por parte de Intel en el año 2015 y un marcado incremento en la implementación de FPGA en entornos de centros de datos, particularmente impulsada por Microsoft (Feldman, 2016).

En la última década, las GPUs de propósito general (GPGPU), lideradas por NVIDIA y su arquitectura CUDA, han transformado el panorama de la computación de alto rendimiento. A pesar de que empresas como Altera y Xilinx mostraron inicialmente un interés limitado en llevar las FPGAs a los centros de datos, la situación ha evolucionado de manera significativa en los últimos dos años (Feldman, 2016).

La participación de gigantes de la industria como Intel y Microsoft indica que las FPGAs se utilizarán cada vez más para acelerar una amplia gama de cargas de trabajo en entornos de centros de datos. Estas aplicaciones abarcan áreas como el aprendizaje automático, el cifrado/descifrado, la compresión de datos, la aceleración de redes y la informática científica, entre otras. Este aumento en la apertura de aplicaciones promoverá el desarrollo de herramientas más completas y sólidas para las FPGAs, lo que, a su vez, facilitará su adopción por parte de una audiencia más amplia, incluyendo aquellos que ejecutan aplicaciones de cómputo de alto rendimiento tradicionales (Feldman, 2016).

1.2. Estudio del Procesamiento en Paralelo

En esta sección, se abordarán los conceptos fundamentales necesarios para comprender el procesamiento en paralelo. Para ello, es esencial comenzar por comprender

el concepto de paralelismo y la Ley de Amdahl. Además, se examinarán los diversos tipos de procesamiento en paralelo que se utilizan con mayor frecuencia.

1.2.1. Paralelismo

El paralelismo es un concepto de informática que se basa en un principio simple, “Divida un gran problema en varios pequeños y resuélvalos al mismo tiempo”, que es muy similar a la frase griega de “Divide y vencerás o dividir para reinar”. La computación paralela es el uso de múltiples recursos computacionales para resolver un problema (Bernal et al., 2017).

El paralelismo se entiende como la ejecución simultánea de una misma tarea particionada y adaptada en múltiples procesadores, ya sea una CPU multinúcleo, una GPU, FPGA o algún otro tipo de arquitectura con varios procesadores, con el objetivo de optimizar los resultados y obtener un menor tiempo de ejecución (Weinbach, 2008).

1.2.2. El Factor SpeedUp

El SpeedUp, también llamado aceleración, es una métrica que compara el rendimiento de un sistema multiprocesador con el de un sistema de un solo procesador. En otras palabras, el SpeedUp cuantifica la eficacia de la paralelización de un programa en comparación con su versión secuencial (Rossainz López, 2020).

De acuerdo con (Soto, 2016), la aceleración de un algoritmo se refiere a la velocidad de la implementación en paralelo en comparación con la implementación secuencial. El SpeedUp es una magnitud adimensional.

Para medir la efectividad de un programa paralelizado frente a la versión secuencial, es necesario medir los tiempos de procesamiento de cada uno y con estos, calcular la relación entre los tiempos de ejecución o Speedup:

- ***Tsec*** = Tiempo de ejecución del programa secuencial
- ***Tpar*** = Tiempo de ejecución del programa en paralelo

$$SpeedUp = Tsec / Tpar$$

Considerando que la ejecución en paralelo ha sido realizada utilizando ***P*** procesadores, el SpeedUp ideal se establece claramente en ***P***. La disminución en el rendimiento se hace evidente a medida que el valor real se aleja de este ideal. Sin embargo, antes de emitir un juicio desfavorable basado en una significativa reducción con respecto a ***P***, es necesario considerar diversos factores adicionales. Estos factores comprenden el tiempo invertido en la paralelización del programa, la calidad del código fuente, la naturaleza

inherente del algoritmo implementado y la parte del código que, por su naturaleza, no puede ser paralelizada (Rossainz López, 2020).

1.2.3. Ley de Amdahl

La Ley de Amdahl, también conocida como el argumento de Amdahl, llamada así por el arquitecto informático Gene Amdahl. Esta ley introduce el concepto de cómo determinar la mejora máxima esperada en un sistema global cuando se realiza una mejora en una parte específica del sistema. Esta ley se emplea comúnmente en el ámbito de la computación paralela para prever la velocidad teórica máxima alcanzable al utilizar múltiples procesadores (Njoroge, 2022).

La Ley de Amdahl se utiliza para expresar de manera matemática cómo una mejora en uno o varios componentes, que son utilizados durante un cierto porcentaje del tiempo de ejecución, impacta en el rendimiento general de una computadora (Njoroge, 2022).

Esta ley nos indica que para que una tarea sea paralelizada, esta se puede dividir en dos partes las cuales las podemos identificar como:

- ***1-P***: Una o más porciones que no se pueden paralelizar.
- ***P***: Una o más porciones paralelizables.

Además, en esta ley se emplean diversos términos, como el tiempo total ***T***, que abarca tanto las partes que no pueden paralelizarse como las que sí pueden. De esta manera, se deriva la siguiente ecuación para calcular el tiempo total de ejecución ***T***:

$$T = (1-P)\ T + P\ T$$

Si consideramos la porción del cálculo que no es susceptible de ser dividida en tareas concurrentes como ***f*** y el número de procesadores como ***p***, la ley de Amdahl establece que el factor máximo de aumento de velocidad, denotado como ***S(p)***, se define de la siguiente manera:

$$S(p)= p\ /\ 1+(p-1)\ f$$

1.2.4. Procesamiento en paralelo

Según (BasuMallick, 2022) el procesamiento en paralelo es un método o una técnica de procesar la información en el que trabajan múltiples flujos de cálculos o tareas de

procesamiento de datos a través de varios núcleos o hilos de procesadores que trabajan de manera simultánea.

El concepto de procesamiento paralelo surge como una solución para simplificar el análisis de grandes volúmenes de datos y extraer información significativa de ellos. Áreas como el procesamiento del habla, el procesamiento de imágenes médicas, la bioinformática y otros campos similares afrontan el desafío de analizar gigantescas cantidades de datos complejos (Madiajagan & Raj, 2019).

En la Figura 9 se muestra un ejemplo de como funciona el procesamiento en paralelo, como se divide un problema en partes más pequeñas para enviar cada instrucción a ejecutarse en un core.

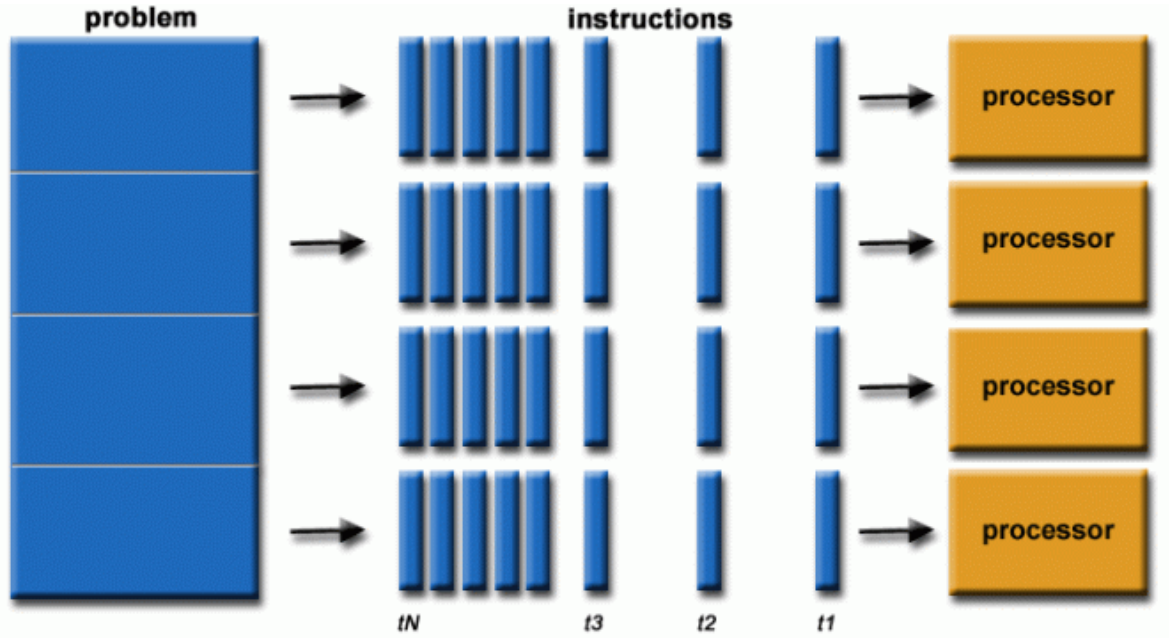


Figura 9. Ejemplo de procesamiento en paralelo

Fuente: (Barney et al., 2018)

1.2.5. Tipos de procesamiento en paralelo

El procesamiento paralelo tiene diferentes formas de clasificarse, entre las que se incluyen MMP, SIMD, MISD, SISD y MIMD, siendo SIMD quizás la más ampliamente reconocida. SIMD, que se refiere a "Single Instruction, Multiple Data" (Instrucción Única, Múltiples Datos), representa un tipo de procesamiento paralelo en el que una computadora dispone de dos o más procesadores que ejecutan el mismo conjunto de instrucciones, pero operan con diferentes tipos de datos (BasuMallick, 2022). A continuación, examinaremos las diversos tipos de procesamiento paralelo y su funcionamiento.

El procesamiento en paralelo puede clasificarse de varias maneras, y una de las taxonomías más reconocidas y utilizadas desde 1966 hasta la fecha es la denominada Taxonomía de Flynn. Esta taxonomía de Flynn se encarga de distinguir las arquitecturas de multiprocesador según dos dimensiones independientes: el flujo de instrucciones y el flujo de datos. Cada una de estas dimensiones puede tener solo dos estados posibles: Único o Múltiple (Barney et al., 2018).

La taxonomía de Flynn clasifica la computación en paralelo en 4 tipos: SISD, SIMD, MISD y MIMD. En la Figura 10 se puede observar que aparte de las 4 clasificaciones de Flynn, existen otras 2 formas, las cuales son SPDM (Single Program, Multiple Data) y MPP (Massively Parallel Processing).

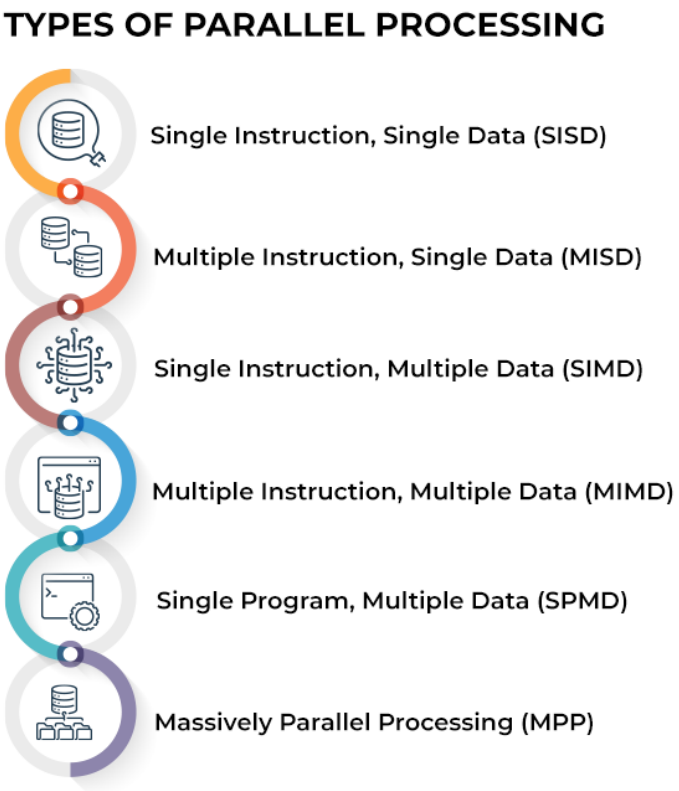


Figura 10. Tipos de procesamiento paralelo

Fuente: (BasuMallick, 2022)

Instrucción Única, Datos Únicos (SISD)

Este tipo de procesamiento recibe su nombre debido a que implica la operación de un solo procesador que maneja un solo algoritmo y una única fuente de datos al mismo

tiempo. SISD, que significa "Single Instruction, Single Data" (Instrucción Única, Datos Únicos), el cual representa un sistema informático el cual, comprende una unidad de control, una unidad de procesamiento y una unidad de almacenamiento, similar a las computadoras seriales convencionales. Las instrucciones se ejecutan de manera secuencial en un sistema SISD, y su capacidad para llevar a cabo procesamiento en paralelo puede variar según la configuración específica del sistema (BasuMallick, 2022).

SISD fue utilizado en las computadoras antiguas, como por ejemplo en minicomputadoras, estaciones de trabajo y PC de y un solo núcleo de generaciones anteriores (Barney et al., 2018). En la Figura 11 se puede observar un gráfico que muestra como es el funcionamiento de Single Instruction, Single Data:

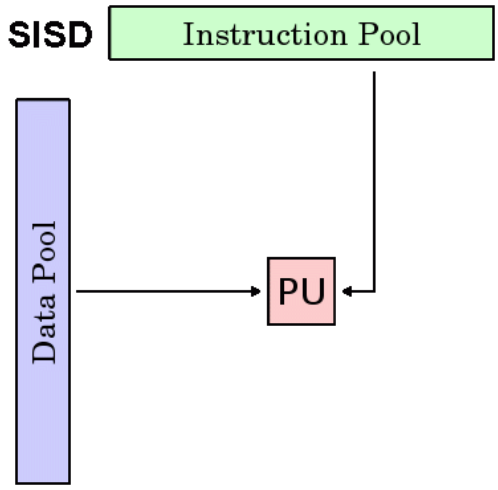


Figura 11. SISD - Single Instruction, Single Data

Fuente: (Barney et al., 2018)

Instrucción única, datos múltiples (SIMD)

Las computadoras que emplean la arquitectura SIMD (Single Instruction, Multiple Data) cuentan con varios procesadores que ejecutan instrucciones idénticas, aunque cada procesador opera con su conjunto único de datos. Esta arquitectura se caracteriza por aplicar el mismo algoritmo a múltiples conjuntos de datos y consta de varios componentes de procesamiento. Todos estos componentes están bajo la supervisión de una sola unidad de control, la cual envía la misma instrucción a cada procesador mientras gestionan distintos datos como se muestra en la Figura 12. Además, esta arquitectura incorpora módulos que facilitan la comunicación simultánea con cada CPU y puede organizarse en segmentos de bits o segmentos de palabras (BasuMallick, 2022).

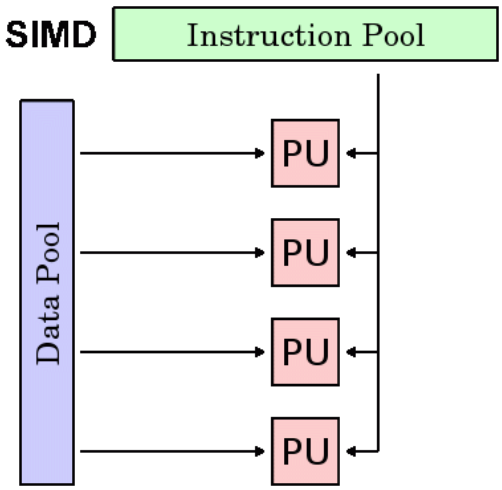


Figura 12. SIMD – Single Instruction, Multiple Data

Fuente: (Barney et al., 2018)

Instrucción múltiple, datos únicos (MISD)

En computadoras que adoptan la arquitectura MISD (Multiple Instruction, Single Data), es común contar con múltiples procesadores. En este enfoque, todos los procesadores comparten los mismos datos de entrada mientras ejecutan diversos algoritmos como se presenta en la Figura 13. Esto permite a las computadoras MISD llevar a cabo simultáneamente múltiples operaciones en el mismo conjunto de datos. Naturalmente, la cantidad de operaciones realizadas está influenciada por la cantidad de procesadores disponibles (BasuMallick, 2022).

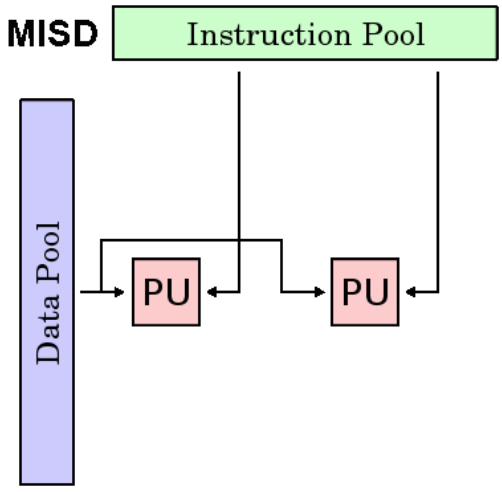


Figura 13. MISD - Multiple Instruction, Single Data

Fuente: (Barney et al., 2018)

Instrucción múltiple, datos múltiples (MIMD)

Las computadoras MIMD (Multiple Instruction, Multiple Data), que se destacan por su capacidad de contar con múltiples procesadores, cada uno de ellos con la habilidad de aceptar su propio flujo de instrucciones de manera independiente como se muestra en la Figura 14. Este tipo de computadoras alberga numerosos procesadores, y cada unidad de procesamiento accede a datos provenientes de flujos de datos distintos. En consecuencia, las computadoras MIMD son capaces de ejecutar múltiples tareas de manera simultánea (BasuMallick, 2022).

A pesar de que las computadoras MIMD ofrecen una mayor versatilidad en comparación con las SIMD o MISD, la creación de algoritmos avanzados para impulsar estas máquinas representa un desafío más considerable. Esto se debe a que todos los flujos de memoria se alteran en el área de datos compartidos, que es accesible por todos los procesadores. En consecuencia, una arquitectura informática MIMD involucra interacciones entre los multiprocesadores (BasuMallick, 2022).

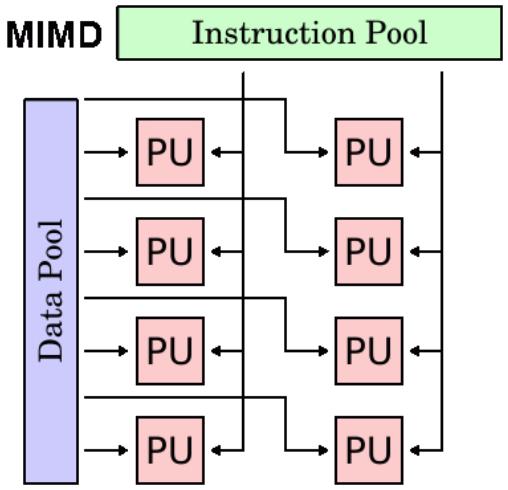


Figura 14. MIMD – Multiple Instruction, Multiple Data

Fuente: (Barney et al., 2018)

Programa único, datos múltiples (SPMD)

Los sistemas SPMD (Single Program, Multiple Data), que se traducen como Programa Único, Datos Múltiples, constituyen un subconjunto de la arquitectura MIMD. Si bien un ordenador SPMD está construido de forma similar a un sistema MIMD, cada uno de sus procesadores es responsable de realizar instrucciones exactamente iguales. Este

enfoque SPMD se utiliza en la programación de paso de mensajes en sistemas de memoria distribuida (BasuMallick, 2022).

En un entorno de memoria distribuida, un conjunto de computadoras individuales, que se denominan colectivamente nodos, se agrupan para formar un ordenador con una memoria compartida. Todos los nodos arrancan su propia aplicación y utilizan funciones de transmisión y recepción de mensajes a fin de comunicarse con los demás nodos. Estos sistemas también pueden utilizar mensajes para coordinar barreras de sincronización. La transferencia de mensajes puede llevarse a cabo mediante diversas técnicas de comunicación, que incluyen protocolos como TCP/IP a través de Ethernet, así como conexiones especializadas de alta velocidad como Supercomputer Interconnect y Myrinet (BasuMallick, 2022).

Procesamiento masivo en paralelo (MPP)

Se ha desarrollado una nueva arquitectura de almacenaje conocida como Procesamiento Masivo en Paralelo (MPP) para administrar la ejecución sincronizada de las distintas operaciones de un determinado programa a través de múltiples procesadores. En este enfoque, cada CPU opera con su propio sistema operativo y memoria, lo que permite la aplicación de este procesamiento coordinado a diversas secciones del programa (BasuMallick, 2022).

Como resultado, las bases de datos que emplean la arquitectura MPP pueden controlar volúmenes masivos de información y realizar un análisis basado en conjuntos de datos grandes notablemente más rápido. Los procesamientos MPP generalmente se comunican entre ellos por medio de una interfaz de mensajería, es posible contar con hasta 200 o más procesos trabajando en una sola aplicación. Esto se logra facilitando la transmisión de mensajes entre procesos mediante un conjunto de enlaces de datos correspondientes (BasuMallick, 2022).

1.3. Programación Heterogénea

La complejidad de las tareas informáticas en la era actual ha experimentado un incremento significativo durante la última década. Diversos aspectos de la informática, como la naturaleza de los datos, las aplicaciones, el entorno de implementación y el hardware, han evolucionado hacia niveles de mayor complejidad. Los sistemas informáticos tradicionales, que se basan en procesadores homogéneos, presentan limitaciones al enfrentar cargas de trabajo complejas y heterogéneas. Diferentes aplicaciones y diversos contextos de implementación imponen requisitos variados en el sistema informático, en

aspectos como la latencia informática, el rendimiento de procesamiento, el consumo de energía, el tamaño del sistema, entre otros, y satisfacerlos se ha vuelto un desafío complejo cuando se utilizan sistemas informáticos homogéneos (Huang et al., 2021).

La computación heterogénea, que combina un CPU y un acelerador, como el GPU que es el más común, constituye el núcleo de la mayoría de los nodos informáticos de alto rendimiento debido a su destacado desempeño y eficiencia energética. Sin embargo, es importante destacar que, en comparación con un FPGA, el GPU requiere un consumo de energía considerablemente mayor. Por esta razón, han surgido sistemas embebidos que emplean FPGA como alternativas para acelerar la computación heterogénea (Valdez & Maldonado, 2021).

Existen varios marcos diseñados para facilitar la programación de sistemas heterogéneos. Uno de ellos es OpenCL, un marco destinado a la programación de sistemas heterogéneos que abarcan CPU, GPU y FPGA. SYCL, por su parte, es un modelo de programación multiplataforma basado en OpenCL y se basa en C++. Además, DPC++ representa la implementación de Intel de SYCL, la cual proporciona una capa de abstracción diseñada para simplificar la computación heterogénea (Figura 15). Un aspecto destacado de estos enfoques es que tanto el host como los aceleradores se programan en el mismo marco, utilizando el lenguaje C/C++ (Huang et al., 2021).

Otra opción es PyLog, que tiene como objetivo simplificar la programación de sistemas heterogéneos. La diferencia radica en que PyLog ofrece una abstracción a un nivel superior, utilizando Python tanto para el host como para los aceleradores, además de proporcionar abstracciones para las interconexiones e interfaces (Huang et al., 2021).

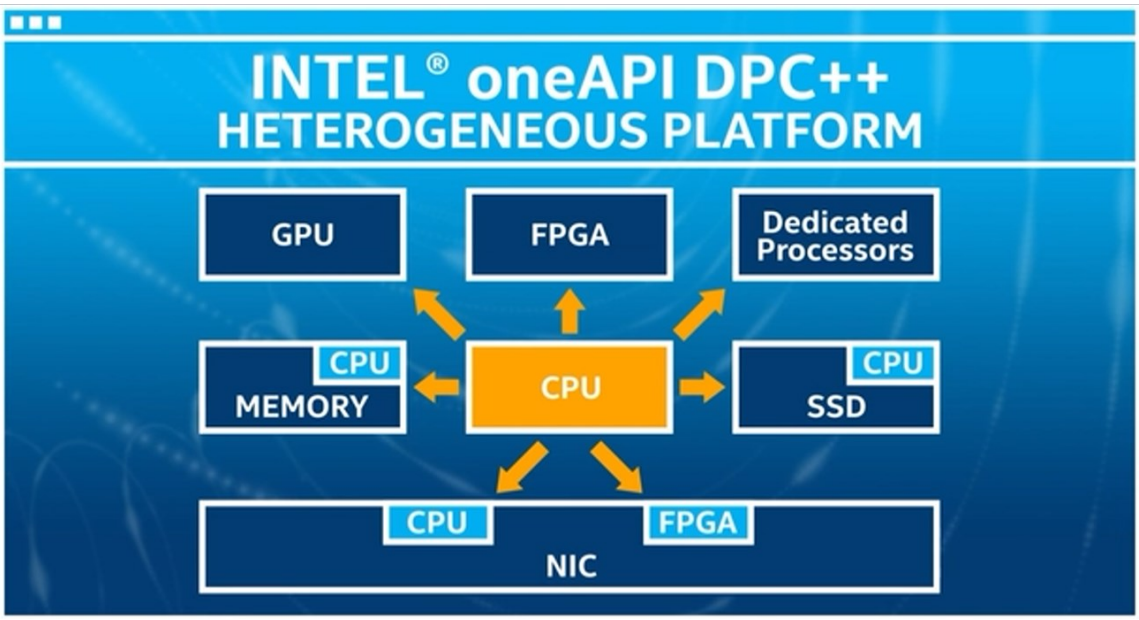


Figura 15. INTEL oneAPI DPC++ - Plataforma Heterogénea

Fuente: (Intel Corporation, 2021)

1.3.1. DPC++

El paralelismo de datos en C++ abre la puerta al acceso de recursos paralelos en sistemas heterogéneos modernos. Con C++, una sola aplicación puede aprovechar una variedad de dispositivos, que van desde GPU, CPU, FPGA hasta circuitos integrados específicos de aplicaciones (ASIC) de IA, según la idoneidad para abordar problemas específicos (Reinders et al., 2021).

SYCL, pronunciado "sickle", es un estándar impulsado por la industria y respaldado por Khronos, que incorpora el paralelismo de datos a C++ para sistemas heterogéneos. Los programas SYCL alcanzan su máximo rendimiento cuando se utilizan con compiladores de C++ compatibles con SYCL, como el compilador de código abierto Data Parallel C++ (DPC++). Importante destacar que SYCL no es un acrónimo, sino simplemente un nombre (Reinders et al., 2021).

El proyecto DPC++, como se indica en la Figura 16, es un compilador de código abierto, nació inicialmente a partir de la iniciativa de empleados de Intel que estaban comprometidos con el sólido soporte al paralelismo de datos en C++. Este compilador DPC++ se basa en SYCL, incluye algunas extensiones y ofrece un amplio respaldo para sistemas heterogéneos que engloban dispositivos CPU, GPU y FPGA. Además de la versión de código abierto de DPC++, existen versiones comerciales disponibles dentro de los kits de herramientas Intel oneAPI. Es relevante mencionar que las funciones implementadas basadas en SYCL son compatibles tanto con las versiones comerciales como con las de código abierto de los compiladores DPC++ (Reinders et al., 2021).

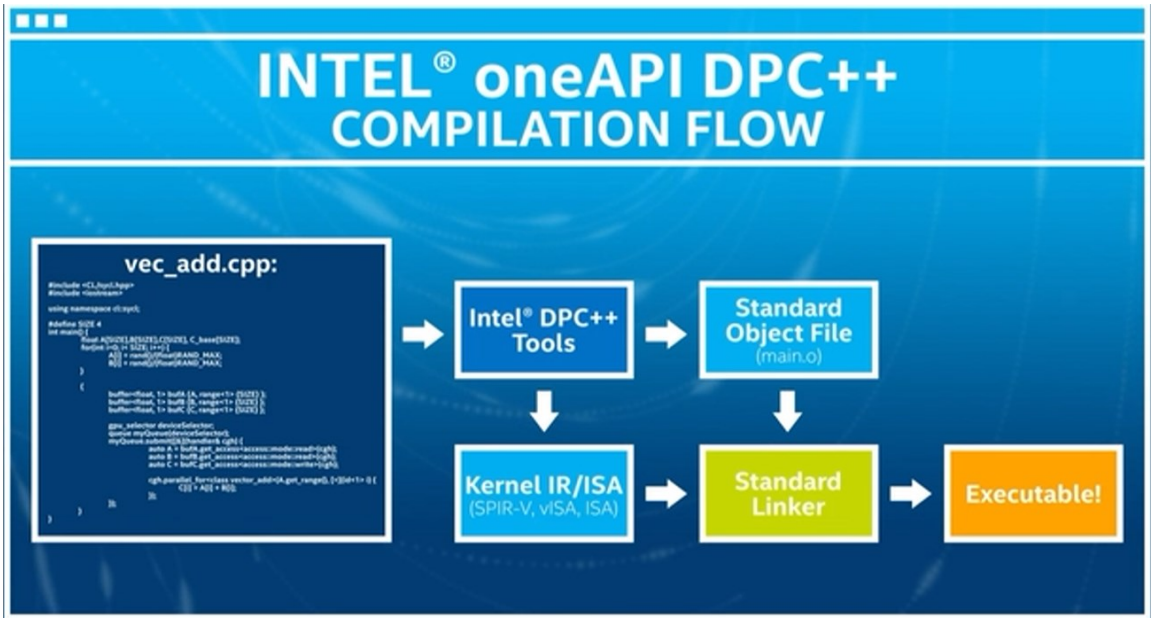


Figura 16. INTEL oneAPI DPC++ - Flujo de compilación

Fuente: (Intel Corporation, 2021)

1.3.2. Queues y Actions

Las colas, conocidas como Queues, representan la única vía mediante la cual una aplicación puede dirigir y gestionar las tareas que un dispositivo debe llevar a cabo. En estas colas, se pueden emplear dos tipos de acciones: (a) código a ejecutar y (b) operaciones de memoria. El código a ejecutar se describe mediante términos como "single_task", "parallel_for", o "parallel_for_work_group". Las operaciones de memoria se utilizan para realizar tareas como la transferencia de datos entre el host y el dispositivo, o para llenar memoria con valores iniciales. El uso de operaciones de memoria es necesario solo cuando se busca un nivel de control más específico del que se ejecuta automáticamente (Reinders et al., 2021).

Es crucial comprender que las colas sirven como el mecanismo principal para controlar un dispositivo y permiten que se asignen y ejecuten acciones específicas en ellas. Importante destacar que las acciones colocadas en una cola no esperan su turno de ejecución de manera inmediata. El host, después de enviar una acción a la cola, continúa con la ejecución del programa, mientras que el dispositivo llevará a cabo la acción solicitada de manera asincrónica y en su propio momento a través de la cola correspondiente (Reinders et al., 2021).

1.4. Algoritmos de alto coste computacional

Existen varios factores que impactan en el coste computacional como el hardware, el sistema operativo, la tamaña de los datos, la complejidad algorítmica, la complejidad computacional, entre otros factores que intervienen en el tiempo de ejecución y en la cantidad de recursos que utiliza para poder ejecutar el programa o algoritmo. Para comprender de mejor manera lo que son los algoritmos de alto coste computacional, primero debemos entender los conceptos de complejidad computacional, complejidad algorítmica y también la clasificación de los algoritmos por su complejidad. Además, es necesario analizar un poco la notación que se utiliza para calcular la complejidad de los algoritmos.

1.4.1. Complejidad computacional

La complejidad computacional es una medida que evalúa la cantidad de recursos informáticos, tanto en términos de tiempo como de espacio, que un algoritmo específico consume durante su ejecución. Los expertos en informática emplean métricas matemáticas de complejidad para anticipar, antes de escribir el código, la velocidad de ejecución de un algoritmo y la cantidad de memoria que requerirá. Estas proyecciones desempeñan un papel fundamental para los programadores, ya que les ayudan a implementar y seleccionar

algoritmos apropiados en aplicaciones del mundo real (Britannica & T. Editors of Encyclopaedia, 2023).

La complejidad computacional abarca un espectro que va desde algoritmos que operan en tiempo lineal, es decir, su rendimiento aumenta de manera proporcional al número de elementos o nodos en la lista, gráfico o red que se está procesando, hasta algoritmos que requieren tiempo cuadrático o incluso exponencial en función de dicho número. En el extremo opuesto de esta escala se hallan los problemas intratables, aquellos cuyas soluciones no pueden resolverse eficazmente. Para abordar estos problemas, los expertos en informática buscan desarrollar algoritmos heurísticos que, aunque no resuelvan el problema de manera exacta, sean capaces de ofrecer soluciones aproximadas en un tiempo razonable (Britannica & T. Editors of Encyclopaedia, 2023).

1.4.2. Complejidad algorítmica

La complejidad de un algoritmo nos proporciona una visión de cómo se comportará dicho algoritmo a medida que incrementemos la cantidad de datos de entrada. Por ejemplo, en el caso de un algoritmo utilizado para ordenar una lista, la complejidad nos brindará información sobre cómo se comportará este algoritmo cuando se le suministren listas de 100,000 elementos o incluso 1 millón de elementos (Moreno, 2023).

La complejidad algorítmica es una medida que evalúa cuánto tiempo requerirá un algoritmo para finalizar su ejecución en función del tamaño de la entrada, denotado como "n". Cuando un algoritmo debe ser escalable, es necesario garantizar que pueda generar resultados en un período de tiempo razonable, incluso cuando se le proporcionan entradas muy grandes. Por esta razón, la complejidad se calcula de manera asintótica, es decir, cuando "n" tiende hacia infinito. Aunque la complejidad suele expresarse en términos de tiempo, en ocasiones también se analiza desde una perspectiva de espacio, lo que implica evaluar los requisitos de memoria del algoritmo (Padmanabhan, 2022).

El análisis de la complejidad de un algoritmo resulta valioso cuando se busca realizar comparaciones entre distintos algoritmos o cuando se intenta mejorar su rendimiento. Esta evaluación de la complejidad algorítmica se encuentra enmarcada en una disciplina de la informática conocida como la teoría de la complejidad computacional. Es relevante destacar que lo que nos interesa principalmente es el orden de complejidad del algoritmo, en lugar del tiempo de ejecución real medido en milisegundos (Padmanabhan, 2022).

1.4.3. Notación Big O

La notación Big-O es el enfoque principal para expresar la complejidad algorítmica. Proporciona un límite superior que señala el rendimiento más desfavorable del algoritmo.

Utilizar esta notación facilita la comparación entre algoritmos, ya que de manera clara indica cómo evoluciona el rendimiento del algoritmo a medida que crece el tamaño de la entrada. A esta característica se le conoce comúnmente como el "orden de crecimiento" (Padmanabhan, 2022).

Cuando se menciona la notación Big-O, es posible hacer referencia a distintos escenarios: el peor, el mejor o el promedio. Por ejemplo, en el algoritmo de búsqueda presentado en el capítulo anterior, que nos permite localizar un elemento en una lista:

- El escenario de peor caso supone que el elemento se encuentra en la última posición.
- En el mejor de los casos, se supone que el ítem se sitúa en la posición inicial.
- El escenario habitual asume que el ítem se halla en la mitad de la búsqueda.

Aunque hay notaciones para expresar tanto el mejor caso como el caso promedio, es más común y conservador hacer referencia al peor caso al hablar de la complejidad de un algoritmo (Moreno, 2023).

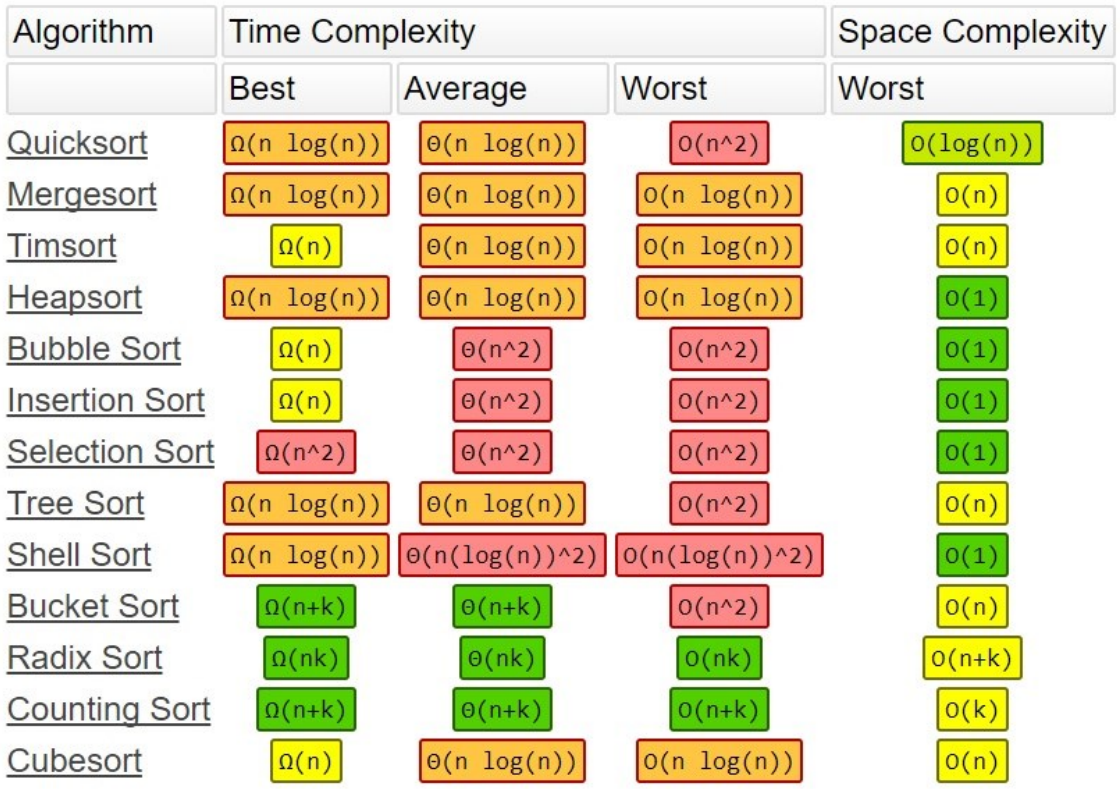
Las estructuras de datos tienen como función principal el almacenamiento de datos, sin embargo, es fundamental considerar la complejidad algorítmica al realizar operaciones con ellas como se muestra en la Figura 17. Por esta razón, es esencial analizar operaciones como la inserción, eliminación, búsqueda e indexación, con el propósito de seleccionar la estructura de datos más apropiada que minimice la complejidad (Requene, 2022).

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Figura 17. Complejidad de operaciones sobre estructuras de datos.

Fuente: (Padmanabhan, 2022)

Entre los algoritmos de clasificación, el más sencillo es posiblemente Bubblesort, pero en términos de eficiencia, en el caso promedio, su complejidad cuadrática lo hace menos eficiente como se indica en la Figura 18. Existen alternativas más eficaces con complejidad logarítmico-lineal, como Quicksort, Mergesort, Heapsort, entre otros. Si la lista ya está ordenada, los algoritmos más eficientes en el mejor de los casos incluyen Bubblesort, Timsort, Insertionsort y Cubesort, todos con un tiempo de ejecución lineal (Padmanabhan, 2022).



Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Figura 18. Complejidad de los algoritmos de ordenamiento

Fuente: (Padmanabhan, 2022)

Complejidad lineal - O(n)

A medida que la cantidad de datos aumenta, el número de operaciones necesarias también aumenta en proporción. Por ejemplo, si se requiere una operación por cada dato, se necesitarán 10 operaciones para 10 datos y 100 operaciones para 100 datos (Moreno, 2023).

La siguiente gráfica Figura 19 la relación entre la cantidad de datos (eje x) y el número de operaciones (eje y):

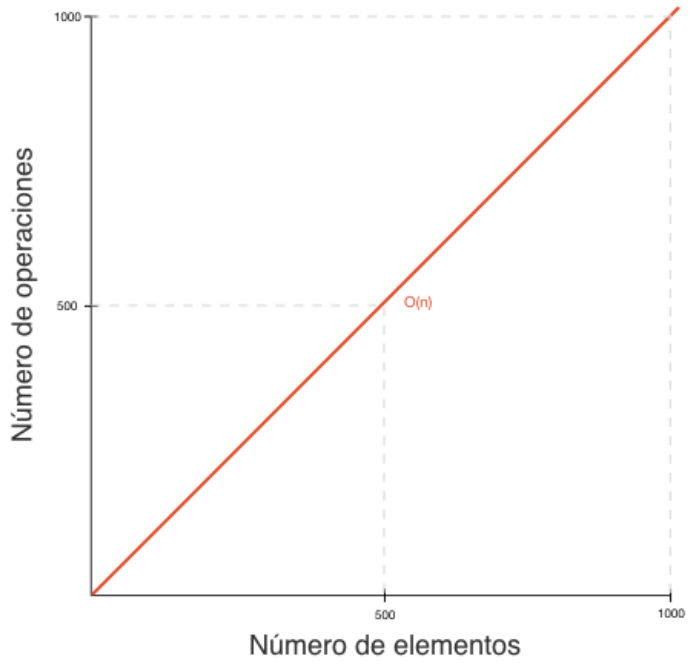


Figura 19. Complejidad lineal

Fuente: (Moreno, 2023)

Complejidad exponencial - O(n^2), O(n^3), etc.

A medida que el número de datos incrementa, la cantidad de instrucciones también aumenta exponencialmente (Figura 20). Por ejemplo, si tenemos una complejidad cuadrática $O(n^2)$ en la que necesitamos 2 operaciones por cada dato, requeriremos 4 operaciones para 2 datos, 9 operaciones para 3 datos, y así sucesivamente.

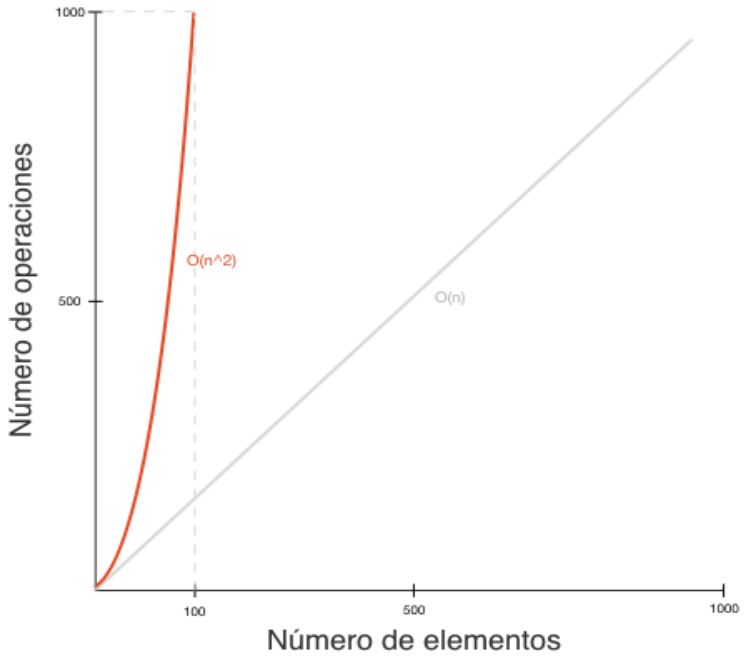


Figura 20. Complejidad exponencial

Fuente: (Moreno, 2023)

Complejidad logarítmica O (log n)

Conforme la cantidad de datos se incrementa, el número de operaciones también se incrementa, pero no de manera proporcional a la cantidad de datos (Moreno, 2023). En la Figura 21 se muestra la curva de crecimiento de la complejidad logarítmica.

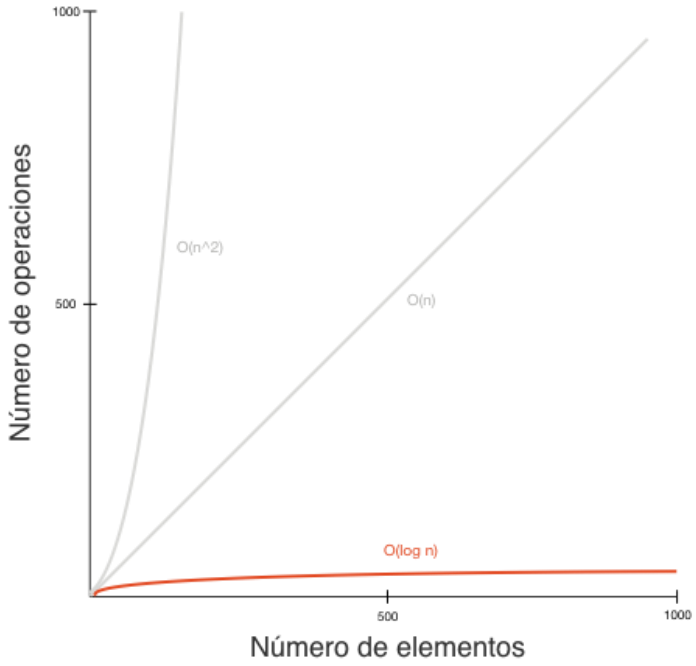


Figura 21. Complejidad logarítmica

Fuente: (Moreno, 2023)

Complejidad constante - O (1)

A medida que la cantidad de datos aumenta, el número de operaciones se mantiene constante.

Cuando se utiliza la notación O (1), significa que se ejecuta una sola operación. En el caso de O (2), se realizan dos operaciones, y así sucesivamente. Sin embargo, lo más relevante no es la cantidad de operaciones en sí, sino que, sin importar cuántos datos de entrada se tengan, el rendimiento permanece constante (Moreno, 2023).

Cada operación puede involucrar acciones como asignar un valor a una variable, realizar una comparación o llevar a cabo cálculos aritméticos, como suma, resta, multiplicación o división, entre otros (Moreno, 2023).

1.5. Buenas prácticas de programación

Los procesadores multinúcleo han estado disponibles durante un largo período de tiempo, pero su adopción generalizada en productos comerciales y su expansión a diversos

segmentos del mercado solo ocurrió recientemente. Esta época de la computación multinúcleo se materializó debido a las limitaciones de potencia y rendimiento que los procesadores de núcleo único alcanzaron. Para lograr mejoras de rendimiento en la era multinúcleo, los desarrolladores se enfrentaron al desafío de realizar cambios significativos en el software actual, transformándolo de aplicaciones secuenciales a aplicaciones paralelas. Esta tarea no es sencilla y plantea nuevos desafíos que abarcan todo el ciclo de desarrollo, desde el análisis y diseño hasta la implementación, depuración y optimización del rendimiento del programa (Embedded Staff, 2013).

Las "mejores prácticas" en el ámbito del software suelen ser el resultado de una combinación de experiencias personales, conocimiento acumulado en la empresa, pensamiento colectivo y, en algunos casos, requisitos específicos del cliente. A medida que los desarrolladores se esfuerzan por crear nuevo software o migrar aplicaciones existentes para aprovechar al máximo las capacidades de procesamiento en paralelo de las arquitecturas multinúcleo, identificar estas mejores prácticas se convierte en un desafío significativo. La Asociación Multinúcleo (MCA) ha abordado este desafío mediante la introducción de la guía de Prácticas de Programación Multinúcleo (MPP) (Evanvzuk, 2013).

1.5.1. Metas de la MPP

La guía de Prácticas de Programación Multinúcleo (MPP) proporciona una colección precisa de recomendaciones clave para el desarrollo de software dirigido a procesadores multinúcleo, aprovechando la tecnología y software ya existentes.

A continuación, se presentan reflexiones sobre las decisiones de alto nivel que se han tomado:

Opciones de API: Se optó por Pthreads y MCAPI por diversas razones. Estas dos API abordan las dos categorías principales de programación multinúcleo: programación paralela de memoria compartida y comunicación a través de mensajes. En primer lugar, Pthreads es ampliamente utilizada para la programación paralela de memoria compartida. MCAPI, por otro lado, es una API relativamente nueva pero innovadora, ya que se centra en la comunicación multinúcleo a través de mensajes, dirigida específicamente a aplicaciones integradas, tanto homogéneas como heterogéneas (The multicore association et al., 2013).

Categorías de Arquitectura: Las categorías de arquitectura, que incluyen multinúcleo homogéneo con memoria compartida, multinúcleo heterogéneo con una combinación de memoria compartida y no compartida, y multinúcleo homogéneo con memoria no compartida, se han organizado en función de su prioridad y de la evaluación del grupo en relación con las aplicaciones integradas. Es importante destacar que la segunda categoría es una expansión de la primera y la tercera. La razón detrás de listar estas tres categorías

por separado radica en la prevalencia y el uso común de técnicas asociadas con la primera y la tercera, es decir, la programación paralela de memoria compartida y el paralelismo a nivel de procesos. Los siguientes párrafos exploran en mayor detalle este enfoque y proporcionan más información al respecto (The multicore association et al., 2013).

1.5.2. Fases de la MPP

La intención es compartir técnicas de desarrollo que se han demostrado efectivas en el contexto de procesadores multinúcleo. Esto conduce a una reducción en los costos de desarrollo, un tiempo de comercialización más corto y un ciclo de desarrollo más eficiente para aquellos que aplican estas técnicas (Embedded Staff, 2013). Las diferentes fases del desarrollo de software que se analizan en la guía MPP, junto con un resumen de cada una, son las siguientes:

- Análisis y Diseño de Alto Nivel del Programa: Esta etapa implica un estudio detallado de la aplicación para identificar las áreas donde se puede introducir concurrencia y desarrollar una estrategia para modificar la aplicación con el fin de admitir la concurrencia (The multicore association et al., 2013).
- Implementación y Diseño de Bajo Nivel: En esta fase se eligen modelos de diseño, los algoritmos y las estructuras de datos apropiados, y se codifica el software con características de capacidad concurrencia (The multicore association et al., 2013).
- Depuración: La depuración se centra en implementar la concurrencia de manera que se minimicen los problemas latentes relacionados con la concurrencia. Esto facilita la detección y resolución de problemas de concurrencia en la aplicación, así como el uso de técnicas para identificarlos (The multicore association et al., 2013).
- Optimización del Rendimiento: En esta etapa, se busca mejorar el tiempo de respuesta y el rendimiento general de la aplicación. Esto se logra identificando y abordando los cuellos de botella que pueden estar relacionados con la comunicación, la sincronización, los bloqueos, el equilibrio de carga y la localización de datos (The multicore association et al., 2013).
- Tecnología Existente: La guía también incluye detalles sobre los modelos de programación y arquitecturas multinúcleo que se utilizan comúnmente en la actualidad (The multicore association et al., 2013).
- Software Existente: El término "software existente" se trata de la utilización actual de la aplicación, que se representa mediante su propio código de software. Los clientes que usan programas existentes prefieren mejorar la aplicación en lugar de una reimplementación completa para habilitar la ejecución en procesadores multinúcleo al desarrollar nuevos productos (The multicore association et al., 2013).

CAPÍTULO 2

Desarrollo

Para la etapa de desarrollo de este proyecto de trabajo de grado fue necesario crear una cuenta en Intel DevCloud y conectarnos a la plataforma mediante la terminal o vía Jupyter Notebook como se detalla en el Anexo A. Una vez creada una cuenta y realizada la conexión a Intel DevCloud, fue necesario asimilar todas las funcionalidades y herramientas que nos proporciona dicha plataforma para el desarrollo de algoritmos con programación heterogénea. Además de configurar la cuenta, se investigó de manera minuciosa la forma de utilizar los recursos de hardware que nos proporciona la herramienta de Intel DevCloud for OneAPI.

En la siguiente fase del desarrollo de este proyecto de grado, se llevó a cabo la investigación y estudio de la programación heterogénea mediante el uso del lenguaje DPC++. Posteriormente, se implementaron algoritmos de elevado coste computacional en el lenguaje Data Parallel C++, aprovechando los conceptos de heterogeneidad y paralelización proporcionados por este lenguaje. El objetivo es demostrar la eficacia de la ejecución de algoritmos de alta complejidad mediante programación heterogénea en comparación con la programación tradicional, para la cual se utilizará el lenguaje de programación C/C++.

2.1. Estudio de la herramienta Intel DevCloud for OneAPI

2.1.1. ¿Qué es Intel DevCloud for OneAPI?

Intel DevCloud es una plataforma de programación en la nube que proporciona Intel, ideada para que desarrolladores, investigadores y estudiantes puedan acceder a una gran variedad de recursos de hardware y software para la experimentación, desarrollo y prueba de diversas aplicaciones de alto rendimiento.

Intel DevCloud está plenamente integrado con Intel oneAPI, una serie de herramientas de desarrollo que facilitan la programación heterogénea. OneAPI cuenta con librerías, compiladores y herramientas de análisis y depuración. DevCloud proporciona acceso a cursos, tutoriales y ejemplos prácticos para ayudar a los usuarios a aprender y aprovechar al máximo la plataforma.

oneAPI es una solución que ofrece un modelo de programación unificado para simplificar el desarrollo en diversas arquitecturas. Incluye un lenguaje unificado y simplificado y bibliotecas para expresar paralelismo y ofrece un rendimiento de lenguaje nativo de alto nivel sin concesiones en una variedad de hardware, incluidas CPU, GPU y

FPGA . La iniciativa oneAPI se basa en estándares de la industria y especificaciones abiertas y es interoperable con los modelos de programación HPC existentes (Intel Corporation, 2021).

A continuación, se presentan algunas funcionalidades importantes que se utilizan en la plataforma de Intel.

2.1.2. Conexión a nodos de computo

Intel DevCloud for oneAPI cuenta con un clúster donde se alojan diferentes arquitecturas de procesadores como CPU, GPU y FPGA. Para conectarnos directamente a un **nodo** con un hardware específico, podemos listar todos los nodos disponibles que cuenten con dicha especificación utilizando el siguiente comando:

```
pbsnodes | grep -B4 <propiedad>
```

Existen diferentes tipos de propiedades por las cuales podemos realizar la filtración de nodos como core, fpga, gpu o xeon. En la Figura 22 se puede observar una lista de nodos filtradas por la propiedad gpu. Es importante tomar en cuenta que para escoger un nodo debemos utilizar solo los nodos en estado **free**, como los nodos s019-n003 y s019-n004.

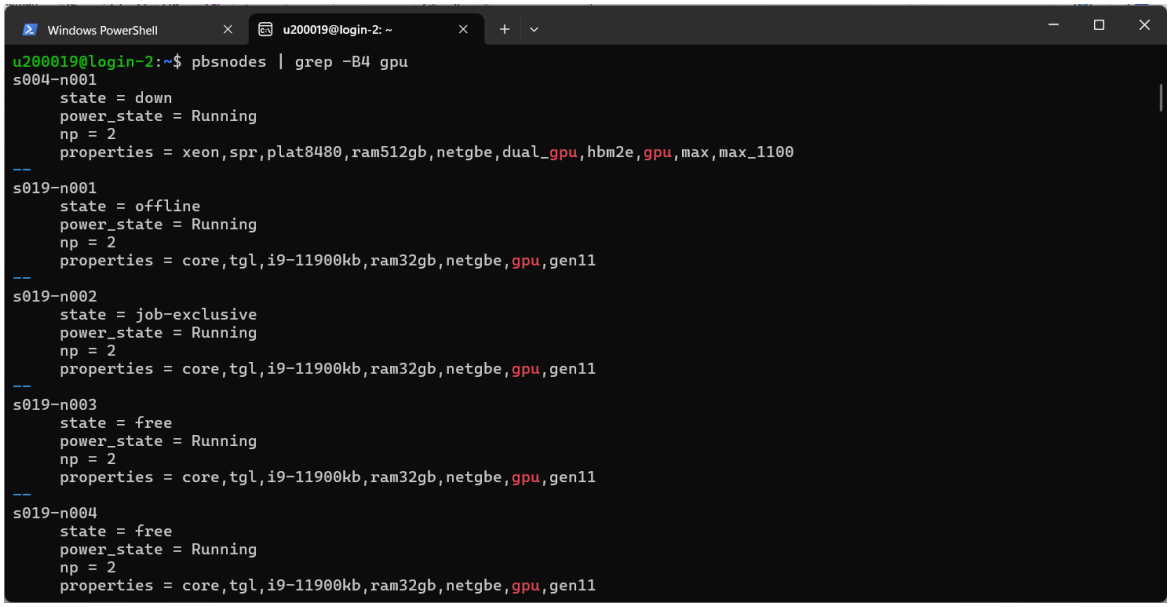


Figura 22. Listar nodos por propiedad

Fuente: Elaboración propia

Para conectarnos de forma interactiva al nodo requerido debemos utilizar el siguiente comando:

```
qsub -I -l nodes=<free_gen9_node>:ppn=2
```

Debemos reemplazar *<free_gen9_node>* con un nodo libre que hayamos identificado, ejemplo:

```
qsub -I -l nodes=s019-n004:ppn=2
```

```
u200019@login-2:~$ qsub -I -l nodes=s019-n004:ppn=2
qsub: waiting for job 2561563.v-qsvr-1.aidevcloud to start
qsub: job 2561563.v-qsvr-1.aidevcloud ready


################################################################################
#      Date:           Fri Jul  5 04:22:35 PM PDT 2024
#    Job ID:           2561563.v-qsvr-1.aidevcloud
#      User:           u200019
# Resources:           cput=75:00:00,neednodes=s019-n004:ppn=2,nodes=s019-n004:ppn=2,walltime=06:00:00
################################################################################

u200019@s019-n004:~$
```

Figura 23. Conexión exitosa a un nodo específico

Fuente: Elaboración propia

También podemos conectarnos a un nodo simplemente utilizando una propiedad específica que necesitemos, como por ejemplo gpu:

```
qsub -I -l nodes=gpu:ppn=2
```

2.1.3. Ejecución de algoritmos

Existen dos formas para ejecutar algoritmos en Intel DevCloud for oneAPI, la primera opción es utilizar la terminal para realizar una **conexión de forma interactiva a un nodo** de cómputo y ejecutar los algoritmos directamente desde la misma terminal utilizando comandos de ejecución como se muestra en la Figura 24. Se puede observar la compilación de un algoritmo en DPC++ y posteriormente la ejecución del programa compilado.

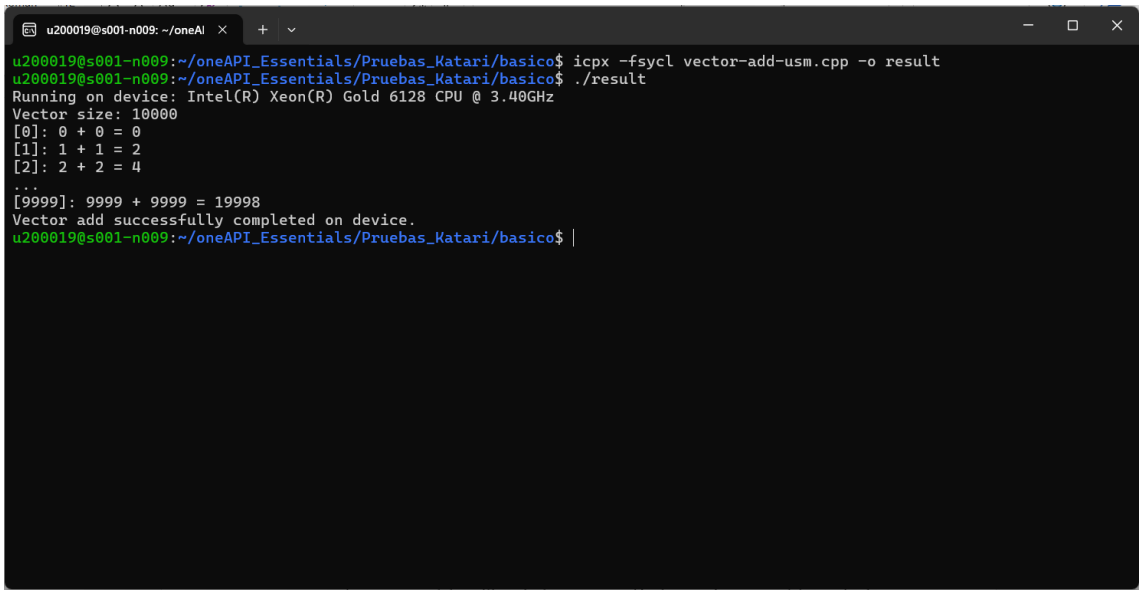


Figura 24. Ejecución de algoritmo DPC++ en un nodo interactivo

Fuete: Elaboración propia

La segunda forma para ejecutar un algoritmo es **enviar un trabajo por lotes**, es decir, que se envía un trabajo a que se ejecute en un nodo libre, sin la necesidad de conectar de forma interactiva a dicho nodo. De esta forma se puede enviar diferentes cargas de trabajo al mismo tiempo y con diferentes especificaciones de hardware. Para enviar un trabajo por lotes es necesario utilizar un script *.sh* donde se detallen los comandos necesarios para ejecutar el algoritmo. A continuación se muestra un script de ejemplo *run.sh* para enviar un trabajo por lotes.

```
1 #!/bin/bash
2 icpx -fsycl basico/vector-add-usm.cpp
3 if [ $? -eq 0 ]; then ./a.out; fi
```

Para enviar el trabajo a un nodo se debe utilizar el siguiente comando en donde *<propiedad>* es la especificación de hardware en la que se va a ejecutar el código.

```
qsub -l nodes=1: <propiedad>:ppn=2 -d . run.sh
```

- **-l nodes=1:<propiedad>:ppn=2** (L minúscula) se utiliza para asignar un nodo completo al trabajo.
- **-d** se utiliza para configurar la carpeta actual como directorio de trabajo para la tarea.
- **run.sh** es el script que se ejecuta en el nodo de cómputo.

Para monitorear los trabajos que se estén ejecutando en ese momoto se utiliza el siguiente comando:

```
watch -n 1 qstat -n -1
```

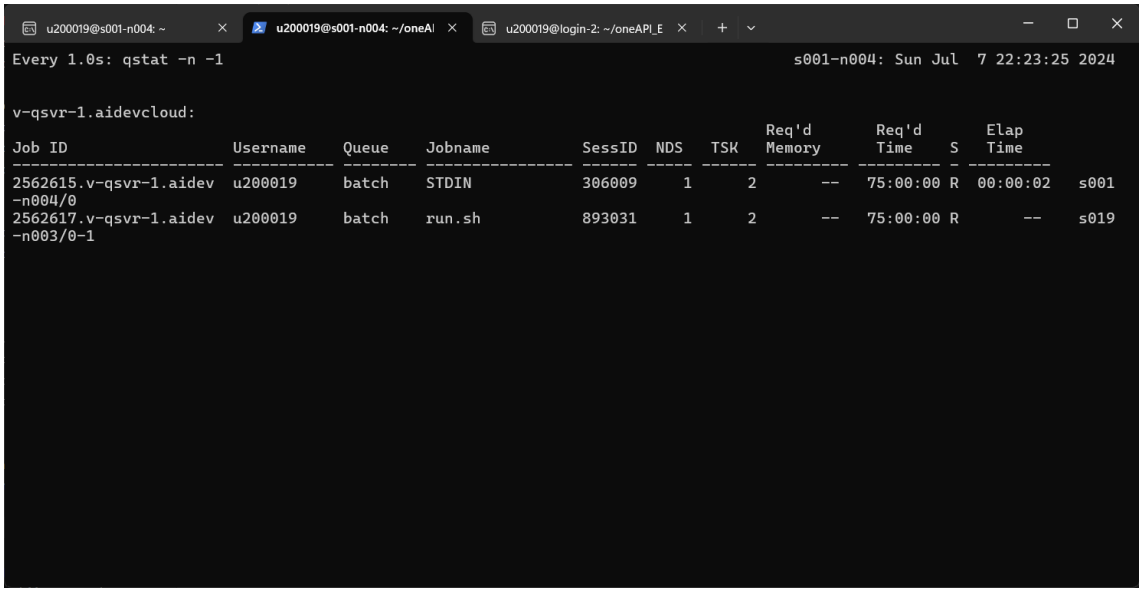


Figura 25. Monitorear los trabajos en Intel DevCloud for oneAPI

Fuente: Elaboración propia

Una vez que el trabajo haya finalizado, desaparecerá de la lista de monitoreo y se crearán dos archivos *run.sh.o******* y *run.sh.e******* donde se encuentran los resultados de la ejecución y los errores en caso el de existir alguno. En la Figura 26 se muestra el contenido del archivo *run.sh.o2562618* utilizando el comando **cat**.

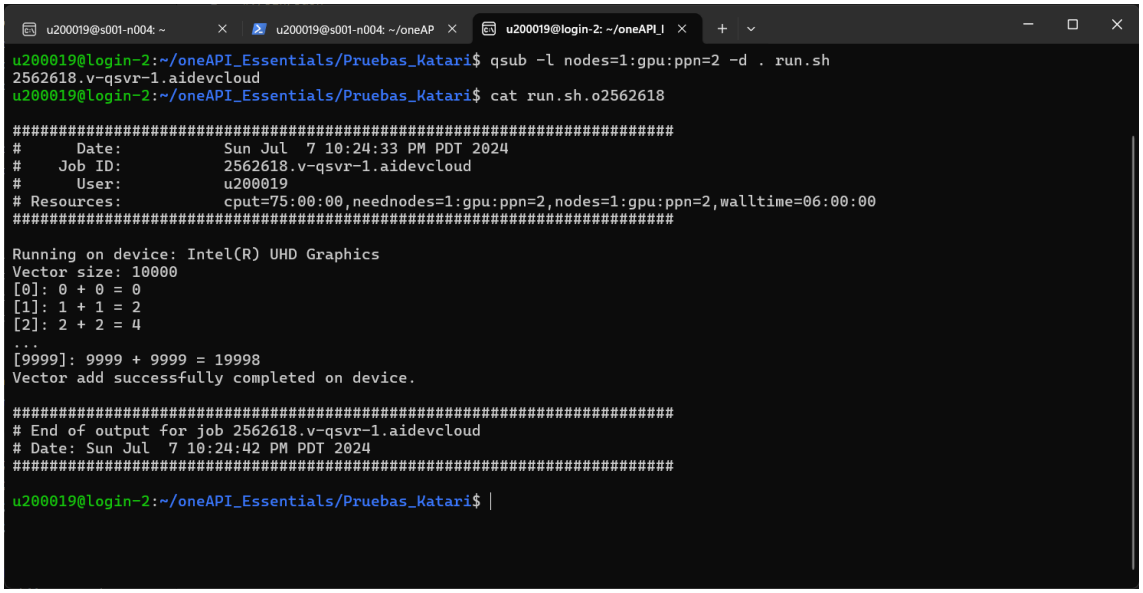


Figura 26. Utilización del comando cat para mostrar contenido del output

Fuente: Elaboración propia

2.2. Estudio del lenguaje de programación heterogénea DPC++

Data Parallel C++ (DPC++) es la implementación del compilador SYCL de oneAPI. Aprovecha los beneficios de productividad modernos de C++ e incorpora el estándar SYCL para paralelismo de datos y programación heterogénea. SYCL es un lenguaje de fuente única donde el código host y los kernels de aceleradores heterogéneos se pueden mezclar en los mismos archivos fuente. Se invoca un programa SYCL en la computadora host y descarga el cálculo a un acelerador. Los programadores utilizan construcciones de biblioteca y C++ familiares con funcionalidades adicionales como una **queue** para la orientación del trabajo, un **buffer** para la gestión de datos y **parallel_for** para el paralelismo para indicar qué partes del cálculo y los datos deben descargarse (Intel Corporation, 2021).

A continuación, se presentan los conceptos fundamentales para comprender el lenguaje de programación heterogénea DPC++, las cuales se utilizaron para el desarrollo de este proyecto.

2.1.1. Cómo compilar y ejecutar un programa SYC(DPC++)

Los tres pasos principales para compilar y ejecutar un programa SYCL son:

1. Inicializar variables de entorno(solo la primera vez)

2. Compile el código fuente de SYCL
3. Ejecute la aplicación

```
source /opt/intel/inteloneapi/setvars.sh

icpx -fsycl simple.cpp -o simple

./simple
```

Figura 27. Compilar y ejecutar un programa DPC++ desde la terminal DevCloud

Fuente: (Intel Corporation, 2021)

2.2.1. Estructura de un programa DPC++

Los programas que utilizan oneAPI requieren la inclusión de **sycl/sycl.hpp**. Se recomienda emplear la declaración *using namespace sycl* para evitar escribir referencias repetidas de sycl al momento de utilizar una clase, función u objeto.

El primer concepto fundamental que tiene este lenguaje de programación es **Device** (Dispositivo), la cual representa las capacidades de los aceleradores en un sistema que utiliza Intel oneAPI Toolkits. La clase **device** contiene funciones miembros para consultar información sobre el dispositivo, lo cual es útil para programas SYCL donde se pueden crear múltiples dispositivos.

- La función *get_info* proporciona información sobre el dispositivo: Nombre, proveedor y versión del dispositivo.
- Los IDs de ítems de trabajo locales y globales
- Ancho para tipos integrados, frecuencia de reloj, ancho y tamaños de caché, en línea o fuera de línea

Selector de dispositivo

Las clases *selector_v* permiten la selección del runtime de un dispositivo particular para ejecutar kernels basados en heurísticas proporcionadas por el usuario. La Figura 28 muestra el uso de los selectores de dispositivos estándar (default_selector_v, cpu_selector_v, gpu_selector_v, accelerator_selector_v) y se la asigna a una variable denominada *q*. Continuando con el código, la siguiente línea imprime el nombre del dispositivo seleccionado en *q*.

Cabe recalcar que el crear una variable *queue* en el main de nuestro programa, es posible utilizarla como parámetro en un método o una función, dependiendo de nuestras

necesidades, pero es importante tomar en cuenta que al emplearla de esta manera se debe utilizar el símbolo “&” para que solo se envié el puntero al objeto, de lo contrario realiza una copia completa del objeto, esto puede ser costoso en términos de tiempo y memoria.

```cpp
#include <sycl/sycl.hpp>

using namespace sycl;
using namespace std;

int main() {
  // Crear una queue de dispositivo con device_selector

  queue q(gpu_selector_v);
  //queue q(cpu_selector_v);
  //queue q(accelerator_selector_v);
  //queue q(default_selector_v);
  //queue q;

  // Imprimir el nombre del dispositivo
  cout << "Device: " << q.get_device().get_info<info::device::name>() << "\n";

  return 0;
}
```

Figura 28. Device selector en DPC++

Fuente: Elaboración propia

El lenguaje de programación DPC++ también nos permite utilizar un **selector de dispositivo personalizado**. Esta funcionalidad nos permite elegir un dispositivo utilizando prioridades dependiendo de diferentes parámetros como pueden ser:

- El nombre del proveedor
- El nombre de un dispositivo GPU específico
- Basado en el tipo de dispositivo

El siguiente código fue extraído de (Intel Corporation, 2021), el cual muestra el Selector de dispositivo personalizado. Busca el dispositivo personalizado específico de un proveedor y si se trata de un dispositivo GPU, en el caso de cumplir con las dos condiciones le da la puntuación más alta (3). La segunda preferencia se da a cualquier dispositivo GPU y la tercera preferencia se da a un dispositivo CPU.

```cpp
#include <sycl/sycl.hpp>
#include <iostream>
using namespace sycl;
```

```
class my_device_selector {
public:
    my_device_selector(std::string vendorName) : vendorName_(vendorName){};
    int operator()(const device& dev) const {
    int rating = 0;

     if (dev.is_gpu() & (dev.get_info<info::device::name>().find(vendorName_)
!= std::string::npos))
        rating = 3;
    else if (dev.is_gpu()) rating = 2;
    else if (dev.is_cpu()) rating = 1;
    return rating;
    };

private:
    std::string vendorName_;
};

int main() {
     //introduzca el nombre del proveedor para el que desea consultar el
dispositivo
    std::string vendor_name = "Intel";
    //std::string vendor_name = "AMD";
    //std::string vendor_name = "Nvidia";
    my_device_selector selector(vendor_name);
    queue q(selector);
    std::cout << "Device: "
    << q.get_device().get_info<info::device::name>() << "\n";
    return 0;
}
```

Queue

La queue o cola en español, envía grupos de comandos para que los ejecute el runtime de SYCL. La queue es un mecanismo mediante el cual el trabajo se envía a un dispositivo como se muestra en la Figura 29. Se pueden asignar una cola a un dispositivo y varias colas al mismo dispositivo.

```
q.submit([&](handler& h) {
  //Codigo de grupos de comando

});
```

Figura 29. Ejemplo simple queue.submit

Fuente: Elaboración propia

Kernel

La clase del kernel encapsula métodos y datos para ejecutar código en el dispositivo cuando se crea una instancia de un grupo de comandos. El programador no construye explícitamente el objeto kernel y se construye cuando se llama a una función de distribución del kernel, como *parallel_for* como se puede apreciar en la Figura 30.

```
q.submit([&](handler& h) {
  h.parallel_for(range<1>(N), [=](id<1> i) {
    A[i] = B[i] + C[i];
  });
});
```

Figura 30. Ejemplo simple de parallel_for

Fuente: Elaboración propia

El trabajo se envía a queues y cada queue está asociada exactamente con un dispositivo como una CPU, GPU o FPGA específica. Se puede decidir a qué dispositivo está asociada una queue y tener tantas queues como sea necesario para distribuir el trabajo en sistemas heterogéneos.

Kernels paralelos – parallel_for

Un Kernel paralelo permite que varias instancias de una operación se ejecuten en paralelo. Esto es útil para descargar la ejecución paralela de un bucle *for* básico, en el que cada iteración es completamente independiente y está en cualquier orden. Los núcleos paralelos se expresan usando la función *parallel_for*.

En la Figura 31 se puede observar la diferencia que existe este ejecutar un buble *for* normal y un bucle utilizando *parallel_for*. Se puede notar que el orden de ejecución del *for* normal, realiza una ejecución secuencial del 0 al 59, mientras que en contraparte la

ejecución del *parallel_for* no sigue un orden específico, sino que va imprimiendo cada iteración de forma independiente a las demás según vaya finalizando la ejecución.

```
Device: Intel(R) UHD Graphics P630
################################################################################
for en paralelo
################################################################################
i: {48} i: {49} i: {32} i: {33} i: {5}  i: {6}  i: {7}  i: {8}  i: {9}  i: {10} i: {11} i: {12} i: {13} i: {14}
i: {15} i: {34} i: {35} i: {36} i: {37} i: {38} i: {39} i: {40} i: {41} i: {42} i: {43} i: {44} i: {45} i: {46}
i: {47} i: {0}  i: {1}  i: {2}  i: {3}  i: {4}  i: {16} i: {17} i: {18} i: {19} i: {20} i: {21} i: {22} i: {23}
i: {24} i: {25} i: {26} i: {27} i: {28} i: {29} i: {30} i: {31}
################################################################################
for en secuencial
################################################################################
i: 0    i: 1    i: 2    i: 3    i: 4    i: 5    i: 6    i: 7    i: 8    i: 9    i: 10   i: 11   i: 12   i: 13 i
: 14    i: 15   i: 16   i: 17   i: 18   i: 19   i: 20   i: 21   i: 22   i: 23   i: 24   i: 25   i: 26   i: 27 i
: 28    i: 29   i: 30   i: 31   i: 32   i: 33   i: 34   i: 35   i: 36   i: 37   i: 38   i: 39   i: 40   i: 41 i
: 42    i: 43   i: 44   i: 45   i: 46   i: 47   i: 48   i: 49
```

Figura 31. Comparativa de orden de ejecución entre for y parallel_for

Fuente: Elaboración propia

2.2.2. Orden de ejecuciones de kernel

La ejecución de un algoritmo en el DPC++ es diferente a la tradicional, mientras que en los lenguajes de programación como C++, Java, Python, entre otros, la ejecución es lineal, es decir, que el programa no continúa a la siguiente línea de código hasta que haya finalizado con las tareas anteriores. En DPC++ los kernel se ejecutan de manera independiente y paralela a las demás.

Si en nuestro programa es necesario que un kernel deba terminal su tarea para que continue con las siguientes, existen dos formas en las que podemos controlar el orden en el que se ejecutan los kernel: de forma implícita y de forma explícita.

En la forma implícita, el mismo lenguaje de programación reconoce si debe esperar a que un kernel finalice una ejecución antes de continuar. La dependencia de datos controla si dos kernels usan el mismo buffer, el segundo kernel debe esperar a que se complete el primer kernel para evitar condiciones de carrera.

En la Figura 32 se puede observar un ejemplo del orden de ejecución de 3 kernels, el primer y segundo kernel se ejecutan una después de la otra por la dependencia de datos, el tercer kernel se ejecuta en paralelo a los demás kernels. Finalmente, el kernel 4 se ejecuta cuando todas las demás instrucciones de hayan completado.

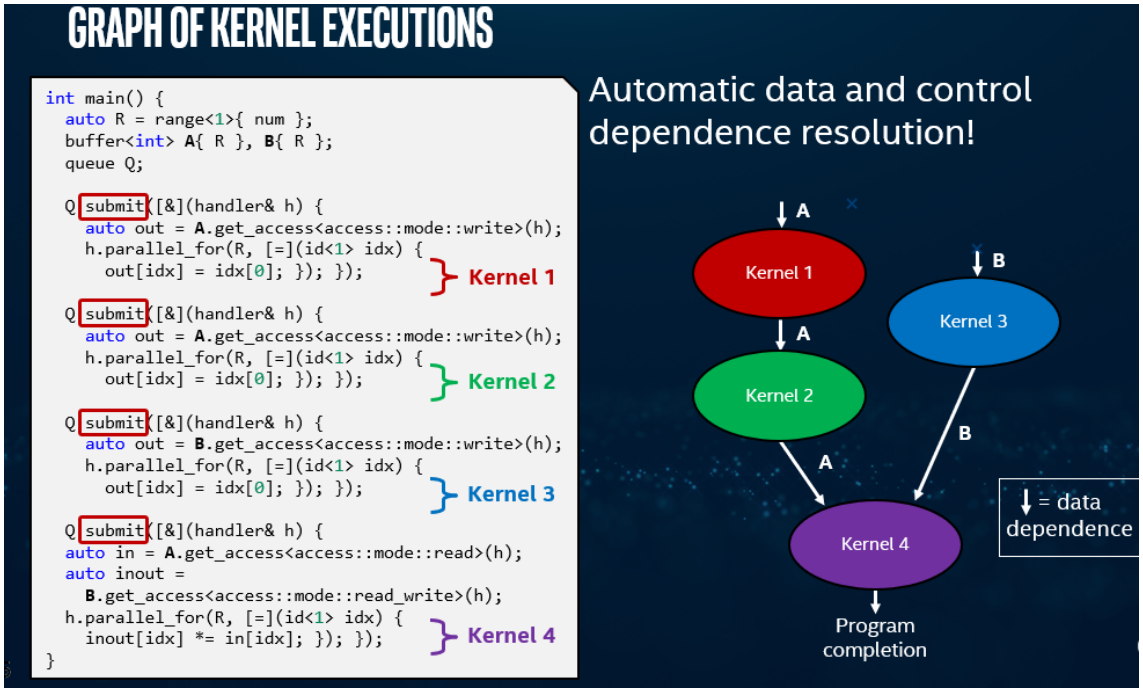


Figura 32. Gráfico de ejecuciones de kernel

Fuente: (Intel Corporation, 2021)

Para poder controlar el orden de ejecución de los kernel de forma explícita se debe utilizar **.wait()** al final del *submit* para indicar al programa que debe de esperar a que termine de ejecutar la instrucción del kernel antes de continuar con las siguientes. En la Figura 33 se puede observar un ejemplo de la utilización del .wait() en un kernel.

```
q.submit([&](handler& h) {
  auto out = stream(1024, 768, h);
  h.parallel_for(range<1>(50), [=](id<1> i) {
    out << "i: " << i << "\t";
  });
}).wait();
```

Figura 33. Utilización de .wait en un kernel

Fuente: Elaboración propia

2.2.3. Manejo de memoria

Un dispositivo, ya sea una CPU, GPU o FPGA diferente al host no puede acceder de manera directa a los datos con los que se está trabajando, para ello existen dos formas por las cuales podemos acceder a los datos desde un dispositivo: buffers y accessors, Unified Shared Memory.

Buffers y accessors

Un **buffer** es un objeto que reserva y administra un bloque de memoria en el host y en el dispositivo. Los buffers son fundamentales en DPC++ para transferir datos entre la

memoria del host y la memoria del dispositivo (GPU, FPGA, etc.). Es importante tomar en cuenta que los buffers tienen un tiempo de vida en el código y se controla mediante llaves "{}", las cuales sirven para especificar de que línea a que línea el buffer está activo.

Un **accessor** es un objeto que proporciona acceso a los datos en un buffer. Es el mecanismo para acceder a los datos del búfer. Los buffers pueden ser datos de 1, 2 o incluso 3 dimensiones. Los accessor se utilizan dentro de un kernel para su ejecución, es necesario especificar como deben ser accedidos los datos: *read_only, write_only, read_write*.

El dispositivo y el host pueden compartir memoria física o tener memorias distintas. Cuando las memorias son distintas, la descarga de cálculo requiere copiar datos entre el host y el dispositivo. SYCL no requiere que el programador administre las copias de datos. Al crear buffers y accessors, SYCL garantiza que los datos estén disponibles para el host y el dispositivo sin ningún esfuerzo del programador (Intel Corporation, 2021).

Unified Shared Memory (USM)

La Memoria Compartida Unificada (USM) es una gestión de memoria basada en **punteros**. USM simplifica el desarrollo para el programador al transferir código C/C++ existente a SYCL. Permite una gestión flexible y directa de la memoria compartida entre el host y el device. USM proporciona un modelo de programación más similar al de CUDA y OpenCL, donde los desarrolladores tienen un control explícito sobre la memoria y pueden usar punteros directamente en el kernel (Reinders et al., 2021).

Ventajas de USM

- **Facilidad de Uso**: Los punteros USM simplifican el código al permitir a los programadores trabajar con punteros directos. Esto hace que el código sea más intuitivo, especialmente para aquellos familiarizados con otros modelos de programación como CUDA.
- **Flexibilidad**: USM proporciona una mayor flexibilidad en la asignación y liberación de memoria en comparación con el modelo de buffers y accessors, permitiendo una gestión de memoria más adaptable.
- **Rendimiento**: Al evitar algunas de las sobrecargas asociadas con la gestión de memoria mediante buffers y accessors, USM puede ofrecer mejoras de rendimiento en determinados escenarios (Reinders et al., 2021).

En la Figura 34 se puede observar el funcionamiento de la Memoria Compartida Unificada. El desarrollador puede hacer referencia a ese mismo objeto de memoria en el código del host y del dispositivo.

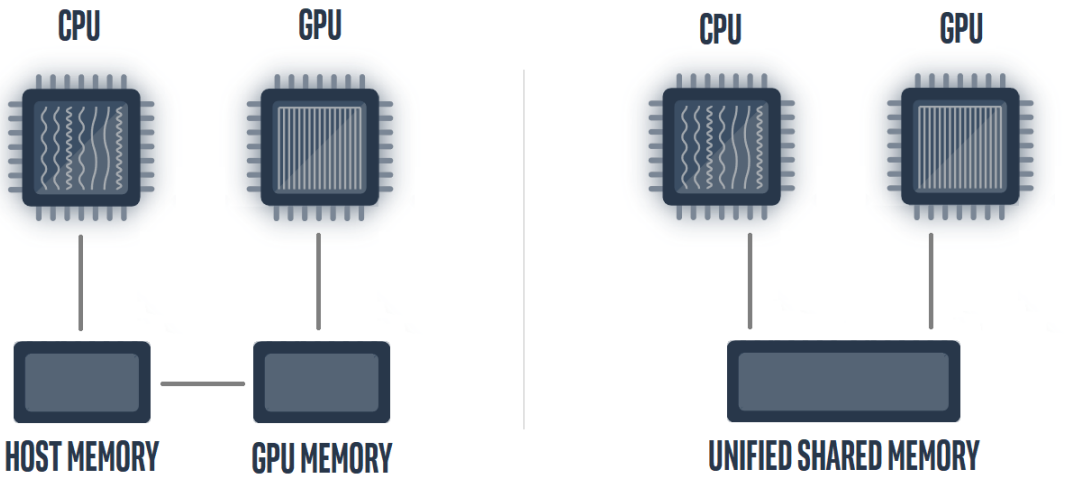


Figura 34. Unified Shared Memory (USM)

Fuente: (Intel Corporation, 2021)

Inicialización de USM: la inicialización a continuación muestra un ejemplo de asignación compartida usando malloc_shared, el parámetro de cola "q" proporciona información sobre el dispositivo con el que se va a compartir la memoria.

```
int *data = malloc_shared<int>(N, q);
```

Es importante que al final del código liberemos la memoria del USM con el siguiente comando:

```
free(data, q);
```

2.2.4. Subgrupos en la ejecución del kernel (ND-Range)

En muchas plataformas de hardware modernas, un subconjunto de **elementos de trabajo**(work-items) de un **grupo de trabajo**(work-group) se ejecuta simultáneamente o con garantías de programación adicionales. Estos subconjuntos de elementos de trabajo se denominan **subgrupos**. Aprovechar los subgrupos ayudará a asignar la ejecución al hardware de bajo nivel y puede ayudar a lograr un mayor rendimiento.

La ejecución paralela con el kernel **nd_range** ayuda a agrupar elementos de trabajo que se asignan a recursos de hardware. Esto ayuda a ajustar el rendimiento de las aplicaciones. El rango de ejecución de un kernel de nd_range se divide en: work-groups, subgroups y work-items como se muestra en la Figura 35.

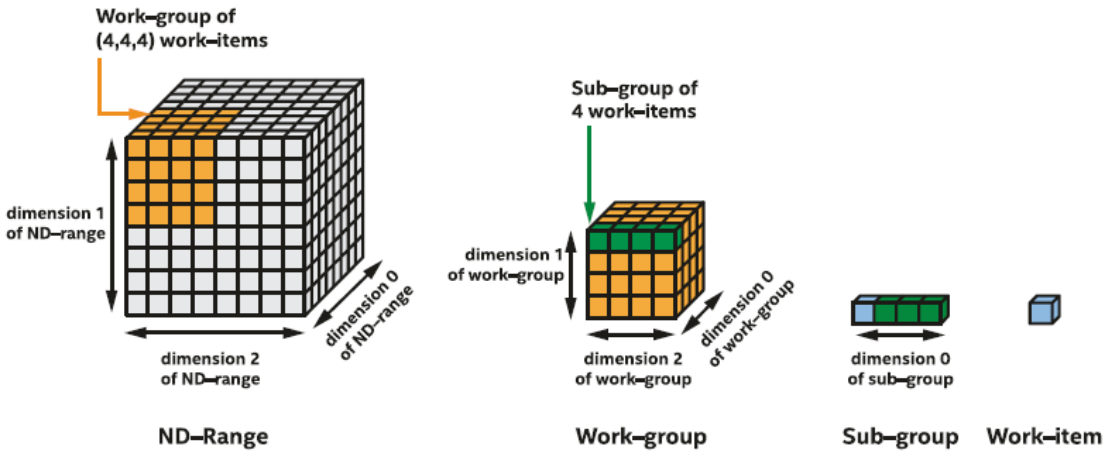


Figura 35. ND_range tridimensional dividido en work-groups, sub-groups y work-items

Fuente: (Reinders et al., 2021)

Al utilizar nd_range y manejar correctamente los subgrupos, los work-items de un subgrupo pueden comunicarse directamente entre ellas, sin necesidad de operaciones de memoria explícitas. También los work-items se pueden sincronizar mediante barreras de subgrupo y garantizar la coherencia de la memoria. Los work-items de un subgrupo tienen acceso a funciones y algoritmos del subgrupo, lo que proporciona implementaciones rápidas de patrones paralelos comunes (Intel Corporation, 2021).

El identificador del subgrupo se puede obtener del nd_item usando *get_sub_group()*:

- sycl::sub_group sg = nd_item.get_sub_group();
- auto sg = nd_item.get_sub_group();

Se puede consultar el identificador del subgrupo para obtener otra información, como la cantidad de elementos de trabajo en el subgrupo, o la cantidad de subgrupos en un grupo de trabajo que serán necesarios para que los desarrolladores implementen el código del kernel usando subgrupos:

- *get_local_id()* devuelve el índice del elemento de trabajo dentro de su subgrupo
- *get_local_range()* devuelve el tamaño del sub_grupo
- *get_group_id()* devuelve el índice del subgrupo
- *get_group_range()* devuelve el número de subgrupos dentro del grupo de trabajo principal

En la Figura 36 se muestra un fragmento de código extraído de (Intel Corporation, 2021) el cual imprime la información de un subgrupo que tiene un tamaño global de 64 y se divide en 4 subgrupos de 16.

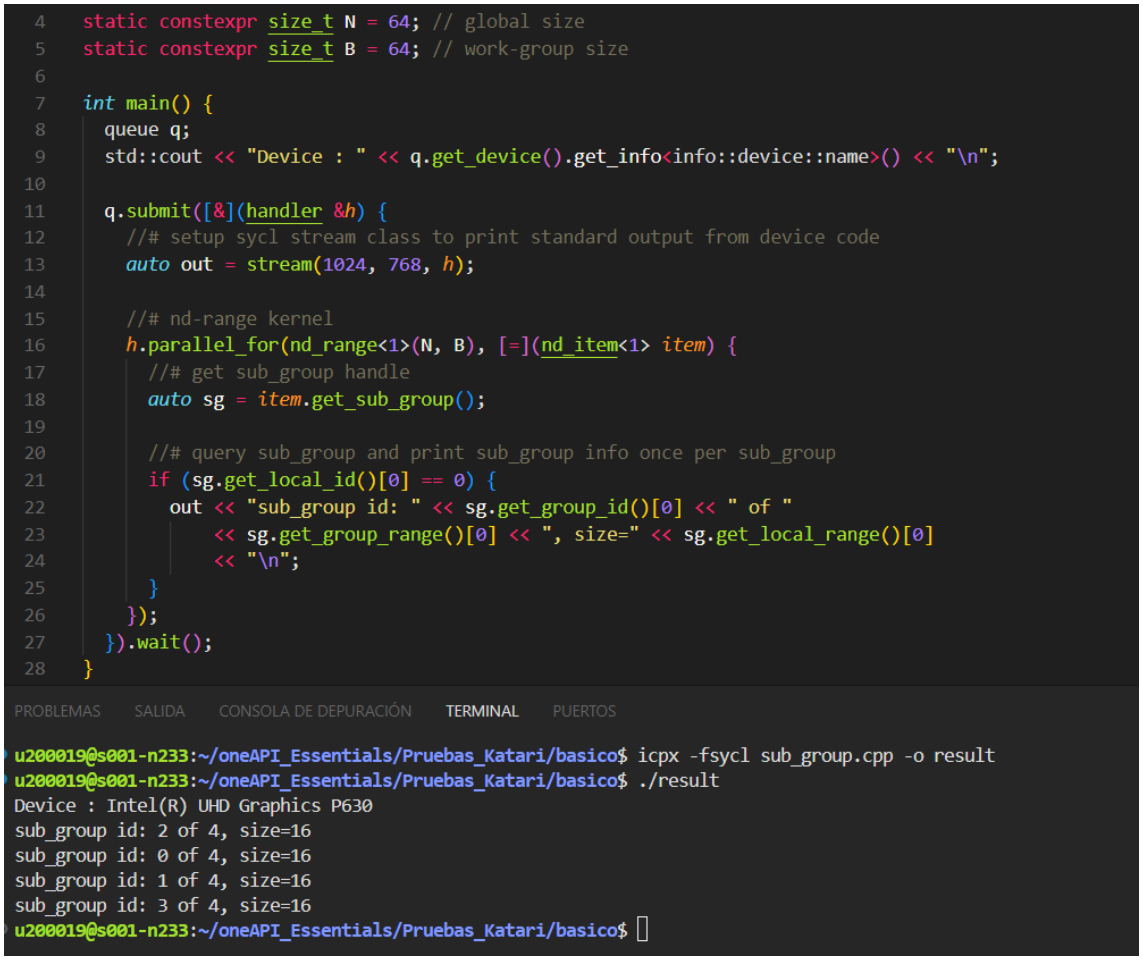


Figura 36. Información del subgrupo

Fuente: Elaboración propia

Para obtener un mejor rendimiento de las aplicaciones, debemos conocer el tamaño específico de subgrupo que permite nuestro dispositivo, la GPU Intel(R) admite tamaños de subgrupos de 8, 16 y 32. De forma predeterminada, la implementación del compilador elegirá el tamaño de subgrupo óptimo, pero también se puede obligar a utilizar un valor específico. El siguiente código muestra como imprimir los tamaños de subgrupos admitidos del dispositivo:

```
auto sg_sizes = q.get_device().get_info<info::device::sub_group_sizes>();
for (int i=0; i<sg_sizes.size(); i++){
        std::cout << sg_sizes[i] << " "; std::cout << "\n";
}
```

2.2.5. Ejemplo básico DPC++: Vector add

A continuación, se muestra un ejemplo básico de un algoritmo utilizando el lenguaje de programación DPC++. El siguiente algoritmo fue adaptado de (Intel Corporation, 2021).

Vector Add es un algoritmo de suma de vectores, es el equivalente a un ¡Hola, mundo! para programas en DPC++. La creación y ejecución del ejemplo verifica que el

entorno de desarrollo esté configurado correctamente y demuestra el uso de las funciones principales de SYCL. Este ejemplo se ejecuta tanto en CPU como en GPU (o FPGA). El algoritmo se ejecuta tanto en la CPU de forma secuencial como en el dispositivo de forma paralela y luego las compara para comprobar que en los dos casos llegan exactamente al mismo resultado.

```
#include <sycl/sycl.hpp>
#include <array>
#include <iostream>
#include <string>

using namespace sycl;

// Array size for this example.
size_t array_size = 10000;

void VectorAdd(queue &q, const int *a, const int *b, int *sum, size_t size) {
  range<1> num_items{size};
  auto e = q.parallel_for(num_items, [=](auto i) { sum[i] = a[i] + b[i]; });
  e.wait();
}


void InitializeArray(int *a, size_t size) {
  for (size_t i = 0; i < size; i++) a[i] = i;
}


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

  try {
    queue q(gpu_selector_v);
    //queue q(cpu_selector_v);
    //queue q(accelerator_selector_v);
    //queue q(default_selector_v);
    //queue q;

    std::cout << "Running on device: "
              << q.get_device().get_info<info::device::name>() << "\n";
    std::cout << "Vector size: " << array_size << "\n";


    int *a = malloc_shared<int>(array_size, q);
    int *b = malloc_shared<int>(array_size, q);
    int *sum_sequential = malloc_shared<int>(array_size, q);
    int *sum_parallel = malloc_shared<int>(array_size, q);

    if ((a == nullptr) || (b == nullptr) || (sum_sequential == nullptr) ||
```

```
      (sum_parallel == nullptr)) {
    if (a != nullptr) free(a, q);
    if (b != nullptr) free(b, q);
    if (sum_sequential != nullptr) free(sum_sequential, q);
    if (sum_parallel != nullptr) free(sum_parallel, q);

    std::cout << "Shared memory allocation failure.\n";
    return -1;
  }

  InitializeArray(a, array_size);
  InitializeArray(b, array_size);

  // Compute the sum of two arrays in sequential for validation.
  for (size_t i = 0; i < array_size; i++) sum_sequential[i] = a[i] + b[i];

  // Vector addition in SYCL.
  VectorAdd(q, a, b, sum_parallel, array_size);

  // Verify that the two arrays are equal.
  for (size_t i = 0; i < array_size; i++) {
    if (sum_parallel[i] != sum_sequential[i]) {
      std::cout << "Vector add failed on device.\n";
      return -1;
    }
  }

  int indices[]{0, 1, 2, (static_cast<int>(array_size) - 1)};
  constexpr size_t indices_size = sizeof(indices) / sizeof(int);

  // Print out the result of vector add.
  for (int i = 0; i < indices_size; i++) {
    int j = indices[i];
    if (i == indices_size - 1) std::cout << "...\n";
    std::cout << "[" << j << "]: " << j << " + " << j << " = "
              << sum_sequential[j] << "\n";
  }

  free(a, q);
  free(b, q);
  free(sum_sequential, q);
  free(sum_parallel, q);
} catch (exception const &e) {
  std::cout << "An exception is caught while adding two vectors.\n";
  std::terminate();
}

std::cout << "Vector add successfully completed on device.\n";
```

```
  return 0;
}
```

La respuesta del algoritmo es imprime en consola, en el cual podemos observar el dispositivo en el que se ejecutó la instrucción, a continuación el tamaño del vector después imprime los 3 primeros valores y el último valor del arreglo resultante, finalmente imprime un mensaje de que se ejecutó exitosamente el algoritmo como se puede apreciar en la Figura 37.

```
u200019@s019-n012:~/oneAPI_Essentials/Pruebas_Katari/basico$ icpx -fsycl vector_add.cpp -o result
u200019@s019-n012:~/oneAPI_Essentials/Pruebas_Katari/basico$ ./result
Running on device: Intel(R) UHD Graphics
Vector size: 10000
[0]: 0 + 0 = 0
[1]: 1 + 1 = 2
[2]: 2 + 2 = 4
...
[9999]: 9999 + 9999 = 19998
Vector add successfully completed on device.
```

Figura 37. Resultado en consola del algoritmo Vector add

Fuente: Elaboración propia

2.3. Selección de algoritmos de alto coste computacional

Para la selección de algoritmos de alto coste computacional que se implementaron en el proyecto de tesis, se tomaron en cuenta los tiempos de ejecución y la complejidad algorítmica de los algoritmos. Posteriormente se optimizó el procesamiento en paralelo utilizando la programación heterogénea con el lenguaje de programación DPC++.

La selección de algoritmos con un alto costo computacional se llevó a cabo en tres etapas. En primera etapa se analizó la complejidad algorítmica de los algoritmos utilizando la notación Big O. En la segunda etapa se calculó los tiempos de ejecución de los algoritmos en un lenguaje de programación tradicional (C++). Por último, se aplicó técnicas de optimización utilizando un lenguaje de programación heterogénea (DPC++) para evaluar el rendimiento de los algoritmos.

El primer algoritmo desarrollado es el de **Distancia Euclidiana**, la cual se implementó en los lenguajes de programación C++ y Data Parallel C++. Se tomó como referencia el tiempo de ejecución obtenido con el lenguaje C++ para realizar la comparativa con el tiempo de ejecución utilizando el lenguaje de programación heterogénea.

El segundo algoritmo implementado es **Floyd Warshall** y se ejecutó de forma secuencial y paralela para realizar la comparativa de rendimiento en los dos casos. En el caso del procesamiento en paralelo utilizando el lenguaje de programación heterogénea DPC++, se ejecutará el algoritmo optimizado en tres distintos tipos de procesadores: Unidad

Central de Procesamiento (CPU), Unidad de Procesamiento Gráfico (GPU) y Matriz de Puertas Programable en Campo (FPGA).

2.4. Algoritmo de Matriz de Distancia Euclidiana (MDE)

La distancia euclidiana o también conocida como distancia euclídea es una distancia ordinaria, es decir, que se puede medir con una regla. Representa la distancia que existe entre dos puntos en un espacio euclidiano. La fórmula de distancia euclidiana es deducida del teorema de Pitágoras (Hernández Baez, 2015).

El algoritmo de distancia euclidiana se basa en una fórmula matemática para calcular la distancia directa entre dos puntos en un espacio de n-dimensiones se expresa de la siguiente forma (Graph Everywhere, 2019).

$$\mathrm{d}_E(\mathrm{p},\mathrm{q}) = \sqrt{(\mathrm{p}_1 - \mathrm{q}_1)^2 + (\mathrm{p}_2 - \mathrm{q}_2)^2 + \cdots + (\mathrm{p}_n - \mathrm{q}_n)^2}$$

La fórmula anterior también la podemos expresar de la siguiente manera:

$$\mathrm{d}_E(\mathrm{p},\mathrm{q}) = \sqrt{\sum_{i=1}^{n} \left(p_i - q_i\right)^2}$$

La Matriz de Distancia Euclidiana (MDE) representa las distancias euclidianas entre todos los pares de puntos en un conjunto de datos. La matriz resultante es una matriz simétrica, es decir, que es una matriz cuadrada (n x n), dist (i, j) = dist (j, i)). Además la diagonal principal donde i = j de la matriz de distancias contiene valores de 0 ya que no existen distancia de un punto a sí mismo (Demey et al., 2011).

La complejidad algorítmica del algoritmo de matriz de distancia euclidiana calculada utilizando la notación Big O, se ha determinado como $O(n^3)$. La presente afirmación indica que el algoritmo de MDE representa una complejidad significativa, dado que su complejidad aumenta exponencialmente con el valor de n. Es importante destacar que $O(n^3)$ es uno de los niveles de complejidad más elevados en la notación Big O.

2.4.1. MDE en C++

El algoritmo de Matriz de Distancia Euclidiana en forma secuencial, utilizando el lenguaje de programación de C++ (Figura 38), está conformado por 3 bucles *for* para realizar el recorrido por todos los puntos de la matriz *data* y calcular la distancia euclidiana entre todos los puntos.

La función de la MDE recibe los siguientes parámetros:

- **data**: Matriz de tipo double de tamaño NxM. Es la matriz original de donde se sustraen los datos.
- **num_rows**: Número entero. Contiene el número de filas de la matriz original de datos *data*.
- **num_colums**: Número entero. Contiene el número de columnas de la matriz original de datos *data*.
- **distances**: Matriz de tipo double de tamaño NxN. Es el puntero a una matriz vacía donde se guardarán los cálculos de la matriz de distancia Euclidiana.

En la función definimos tres variables de tipo entero (i, j, k) las cuales funcionarán como contadores para los tres ciclos *for* que tenemos a continuación. Además, hemos declarado la variable temp de tipo double en la cual almacenaremos el cálculo de la distancia Euclidiana en cada iteración.

A continuación, inicializamos tres bucles de tipo *for*, el primer y segundo *for* realizan el ciclo de 0 a *num_rows* (número de filas), esto es necesario para recorrer una matriz de tamaño NxN la cual es la matriz resultante. Adentro de los bucles, inicializamos *temp* con un valor de 0 para que cada iteración este valor se reinicie. Dentro de los dos iteradores se encuentra un tercer iterador el cual hace el recorrido de todas las columnas de datos.

Finalmente, dentro de los tres bucles *for* codificamos el algoritmo para calcular la distancia euclidiana al cuadrado. El algoritmo realiza la resta entre dos puntos de la matriz de datos elevado cuadrado, guardamos el resultado en la *variable* temp. Una vez finalizada el ciclo que pasa por las columnas, guardamos el resultado en la matriz *distances* en la posición (i, j). Al momento de guardar el resultado, se aplica la raíz cuadrada al resultado debido a que anteriormente el algoritmo codificado calcula la distancia euclidiana al cuadrado.

```
static int euclidean_distances(double** data, int num_rows, int num_columns, double** distances){
    int i,j,k;
    double temp;
    for (i = 0; i < num_rows; i++){
        for (j = 0; j < num_rows; j++){
            temp = 0.0;
            for (k = 0; k < num_columns; k++){
                temp += (double) pow(data[i][k] - data[j][k], 2);
            }
            distances[i][j] = sqrt(temp);
        }
    }
    return 0;
}
```

Figura 38. Algoritmo de la MDE en C++

Fuente: Elaboración Propia

2.4.2. MDE en DPC++

Para el desarrollo del algoritmo de la MDE en el lenguaje de programación heterogénea DPC++ (Figura 39), es indispensable realizar unos cuantos pasos previos antes de comenzar a calcular la distancia euclidiana. La función de la MDE recibe cuatro parámetros, un vector *data* que contiene el dataset, *num_rows* y *num_columns* en donde se almacena el número de filas y el número de columnas que tiene la matriz de datos, finalmente un vector denominado *distances* donde se guardará la matriz resultante o matriz de distancias euclidianas.

El primer paso es crear una cola denominada *queue* y definimos es selector, en el código de ejemplo se utilizó el gpu_selector_v, posteriormente de imprime el nombre del dispositivo seleccionado. También se define una variable *num_items* que se utiliza para definir el rango de unidimensional de iteraciones.

A continuación, creamos los buffers *bufferData* y *bufferDistances* para los vectores que recibimos como parámetros *data* y *distances* respectivamente.

buffer bufferData(data) es una forma más directa y conveniente cuando ya tienes un vector con datos listos para usar en el buffer.

buffer bufferDistances(distances.data(), num_items) en esta forma de inicializar un buffer se apunta a los datos presentes en la dirección de memoria **distances.data()** que en este caso es un vector previamente inicializado pero sin ningún dato almacenado.

Para enviar la carga de trabajo al kernel se utiliza *queue.submit*, dentro de esta creamos los respectivos accessors accessorData y accessorDistances para poder acceder a los datos de buffer desde el dispositivo.

Posteriormente se implementa un *parallel_for* de dos dimensiones de rango *num_rows x num_rows* para recorrer todos los puntos de la matriz de *Distances*. Dentro del buble paralelo asignamos a dos variables de tipo entero (i, j) los índices (index[0], index[1]) respectivamente para almacenar la iteración en la que se encuentra la ejecución. Cabe recalcar que se utiliza la fórmula "**i * num_colums + j**" para recorrer las matrices debido a que estas matrices están almacenadas en un vector en forma de un arreglo unidimensional. Además, inicializamos una variable de tipo double denominado *temp* en el que se guardarán los resultados del cálculo de distancia euclidiana también se me asigna un valor de 0.0 para que en cada iteración el valor de temp se reinicie.

Finalmente, se utiliza un ciclo for para recorrer las columnas de la matriz de datos, se realiza el cálculo de la distancia euclidiana al cuadrado y se almacena el resultado en la

variable temp. Al final, se guarda la raíz cuadrada del resultado obtenido utilizando *accessorDistances*.

```
void euclidean_distances(const vector<double> &data, int num_rows, int num_columns, vector<double> &distances){
    queue queue(gpu_selector_v);
    cout << "Device : " << queue.get_device().get_info<info::device::name>() << std::endl;
    range<1> num_items{distances.size()};
    {
        // Crear buffers para que el dispositivo acceda a los matrices
        buffer bufferData(data);
        buffer bufferDistances(distances.data(), num_items);

        queue.submit([&](handler & cgh) {
            accessor accessorData(bufferData, cgh, read_only);
            accessor accessorDistances(bufferDistances, cgh, write_only, no_init);

            cgh.parallel_for(range(num_rows, num_rows), [=](auto index) {
                int i = index[0];
                int j = index[1];
                double temp = 0.0f;
                for (int k = 0; k < num_columns; k++) {
                    temp += (double) pow(accessorData[i * num_columns + k] - accessorData[j * num_columns + k], 2);
                }
                accessorDistances[i * num_rows + j] = sqrt(temp);
            });
        }).wait();
    }
}
```

Figura 39. Algoritmo de la MDE en DPC++

Fuente: Elaboración Propia

Nota: El código que se envía al device o dispositivo, se ejecuta de forma asíncrona al resto del código, es decir una vez se envía esa porción de código al dispositivo se continua con la ejecución del resto del algoritmo. Si es crucial para nuestro programa que la instrucción enviada al dispositivo se complete antes de proceder, es imprescindible emplear ".wait()" para indicar al programa que debe esperar a que dicha instrucción concluya antes de continuar.

2.5. Algoritmo All Pairs Shortest Paths de Floyd Warshall

El problema planteado del algoritmo de todos los pares de caminos más cortos (All Pairs Shortest Paths) se refiere a encontrar los caminos más cortos entre todos los pares de nodos o vértices en un grafo. El algoritmo de Floyd-Warshall ofrece una metodología para abordar este problema al evaluar numerosas rutas potenciales en grafos ponderados, tanto dirigidos como no dirigidos, con el fin de determinar la ruta más corta. (Intel Corporation, 2023).

Se trata de una herramienta computacional esencial en una variedad de casos de uso y aplicaciones del mundo real, ya que facilita la búsqueda de las rutas más cortas en situaciones prácticas comunes, como navegación, juegos, fabricación, comunicaciones e investigación.

Algunos de los ámbitos de aplicación más frecuentes según (Intel Corporation, 2023) incluyen:

- Robots y vehículos autónomos: utilizados para la navegación en entornos con obstáculos.
- Redes informáticas y de telecomunicaciones: para asegurar un enrutamiento eficiente de datos.
- Sistemas de información geográfica (SIG): empleados en servicios cartográficos y de localización.
- Sistemas de transporte: para la optimización de rutas, como las conexiones aéreas o las redes de carreteras.
- Diseño de circuitos: utilizado en la optimización del diseño de circuitos electrónicos.
- Desarrollo de videojuegos: para mejorar los algoritmos de búsqueda de rutas para las entidades de juego.

El algoritmo de Floyd Warshall utiliza un enfoque de programación dinámica, el algoritmo genera una matriz de caminos (distancias) más costas entre todos los pares de nodos en un grafo dirigido o no dirigido (Intel Corporation, 2023).

El algoritmo funciona de la siguiente manera:

- Se inicializa la matriz de resultados como la matriz del grafo de entrada.
- Se consideran todos los nodos uno por uno; se actualiza la ruta más corta entre cada par de nodos, incluido el nodo seleccionado como nodo intermedio.
- La elección de un nodo k como nodo intermedio implica que los nodos {0,1,…,k-1} ya han sido seleccionados como nodos intermedios.
- Suponiendo que el elemento d[i][j] de la matriz denote la distancia más corta desde un nodo origen i a un nodo destino j, para cada par de nodos (i, j), se aplica una de las siguientes condiciones:
    - Si k no es un nodo intermedio en el camino más corto de i a j, entonces d[i][j] permanece sin cambios.
    - De lo contrario, d[i][j] se actualiza a (d[i][k] + d[k][j]), siempre y cuando d[i][j] > (d[i][k] + d[k][j]).

Utilizando la notación Big O para calcular la complejidad algorítmica del algoritmo de Floyd Warshall, se ha determinado una complejidad de $O(n^3)$. La presente afirmación indica que el algoritmo de Floyd Warshall para el problema de All Pairs Shortest Paths representa una complejidad significativa, dado que su complejidad aumenta exponencialmente con el valor de n. Es importante destacar que $O(n^3)$ tiene un nivel de complejidad muy alta según la notación Big O.

2.5.1. Algoritmo All Pairs Shortest Paths en C++

El algoritmo de All Pairs Shortest Paths de forma secuencial (Figura 40), recibe como parámetro un puntero de tipo entero *graph*, el cual hace referencia a un arreglo unidimensional en donde se almacena el grafo en el que se va a trabajar. La función de FloydWarshall cuenta con tres bucles *for*. El primer bucle *for* se utiliza para seleccionar un nodo intermedio “k”, los siguientes dos bucles *for* se utilizan para recorrer cada nodo del arreglo unidimensional de *graph*.

Finalmente, dentro de los 3 bucles *for* se encuentra codificado el algoritmo de Floyd Warshall, utilizando una condicional *if* para comprobar si graph [i][j] > (graph [i][k] + graph [k][j]), en el caso de cumplir la condición se actualiza graph [i][j] = (graph [i][k] + graph [k][j]).

El siguiente fragmento de código fue extraído del repositorio oficial de Intel donde se realiza una comparativa detallada entre al algoritmo de All Pairs Shortest Paths de Floyd Warshall de forma secuencial y de forma paralela (Intel Corporation, 2023).

```
// The basic (sequential) implementation of Floyd Warshall algorithm for
// computing all pairs shortest paths.
void FloydWarshall(int *graph) {
  for (int k = 0; k < nodes; k++) {
    for (int i = 0; i < nodes; i++) {
      for (int j = 0; j < nodes; j++) {
        if (graph[i * nodes + j] >
            graph[i * nodes + k] + graph[k * nodes + j]) {
          graph[i * nodes + j] = graph[i * nodes + k] + graph[k * nodes + j];
        }
      }
    }
  }
}
```

Figura 40. Algoritmo de Floyd Warshall en forma secuencial

Fuente: Elaboración propia

2.5.2. Algoritmo All Pairs Shortest Paths en DPC++

El código de All Pairs Shortest Paths implementa el algoritmo de Floyd Warshall para calcular una matriz que representa la distancia mínima desde cada nodo hacia todos los demás nodos en el grafo. El algoritmo emplea una técnica de procesamiento de bloques paralelos, donde múltiples bloques de código se ejecutan simultáneamente, cada uno encargado de una tarea específica. Para acelerar el rendimiento, los bloques de cálculo intensivo se envían generalmente a la GPU. Además, el código se compila utilizando el compilador Intel oneAPI DPC++/C++. Se aprovechan conceptos de SYCL como kernels,

selector de dispositivos, memoria compartida unificada y grupos de comandos para optimizar el proceso de cálculo.

En la Figura 41 se muestra la implementación del bucle interno del algoritmo de Floyd Warshall en forma de bloques paralelos. La función recibe como parámetro un *nd_item<1>* en donde se referencia un elemento de dimensión 1 en el espacio de trabajo paralelo. Así mimo, recibe como bloques locales *A, B, C* y los índices *i, j*.

El algoritmo trabaja con los bloques locales (LocalBlock),es decir, que el algoritmo fragmenta en varios bloques el trabajo para ejecutar cada una de forma paralela para optimizar la ejecución del algoritmo.

```
void BlockedFloydWarshallCompute(nd_item<1> &item, const LocalBlock &C,
                                 const LocalBlock &A, const LocalBlock &B,
                                 int i, int j) {
  for (int k = 0; k < block_length; k++) {
    if (C[i][j] > A[i][k] + B[k][j]) {
      C[i][j] = A[i][k] + B[k][j];
    }

    item.barrier(access::fence_space::local_space);
  }
}
```

Figura 41. Bucle interno de implementación en bloques paralelos del algoritmo de Floyd Warshall

Fuente: (Intel Corporation, 2023)

En la implementación en bloques paralelos, cada hilo se encarga de procesar una celda específica dentro de un bloque. Para cada bloque, la función mencionada anteriormente se llama repetidamente, una vez por cada celda en el bloque. Cada una de estas llamadas realiza un número de iteraciones igual al número de bloques a lo largo de una sola dimensión. Todos los hilos que trabajan simultáneamente en un bloque se sincronizan al final de cada iteración, garantizando la precisión de los resultados (Intel Corporation, 2023).

La técnica de bloques paralelos consta de tres fases, donde cada fase implementa un bucle en bloques ubicados en posiciones específicas de la matriz de adyacencia del grafo, de la siguiente manera:

- La fase 1 opera en bloques en la diagonal de la matriz.
- La fase 2 opera en bloques que están en la misma fila o columna que un bloque en la diagonal.
- La fase 3 opera en bloques diferentes a los manejados por las fases 1 y 2.

Cada ronda posterior de estas fases se inicia una vez que se completa la ronda anterior, con las siguientes condiciones:

La fase 1 se ejecuta de forma independiente.

La fase 2 se inicia únicamente después de que se completa la fase 1.

La fase 3 se inicia únicamente después de que se completa la fase 2.

El siguiente algoritmo fue extraído del repositorio oficial de Intel donde se realiza una comparativa detallada entre al algoritmo de All Pairs Shortest Paths de Floyd Warshall de forma secuencial y de forma paralela (Intel Corporation, 2023).

```
typedef local_accessor<int, 2>
    LocalBlock;

void BlockedFloydWarshallCompute(nd_item<1> &item, const LocalBlock &C,
                                 const LocalBlock &A, const LocalBlock &B,
                                 int i, int j) {
  for (int k = 0; k < block_length; k++) {
    if (C[i][j] > A[i][k] + B[k][j]) {
      C[i][j] = A[i][k] + B[k][j];
    }

    item.barrier(access::fence_space::local_space);
  }
}

// Phase 1 of blocked Floyd Warshall algorithm. It always operates on a block
// on the diagonal of the adjacency matrix of the graph.
void BlockedFloydWarshallPhase1(queue &q, int *graph, int round) {
  // Each group will process one block.
  constexpr auto blocks = 1;
  // Each item/thread in a group will handle one cell of the block.
  constexpr auto block_size = block_length * block_length;

  q.submit([&](handler &h) {
    LocalBlock block(range<2>(block_length, block_length), h);

    h.parallel_for<class KernelPhase1>(
        nd_range<1>(blocks * block_size, block_size), [=](nd_item<1> item) {
          auto tid = item.get_local_id(0);
          auto i = tid / block_length;
          auto j = tid % block_length;

          // Copy data to local memory.
          block[i][j] = graph[(round * block_length + i) * nodes +
                              (round * block_length + j)];
          item.barrier(access::fence_space::local_space);
```

```
          // Compute.
          BlockedFloydWarshallCompute(item, block, block, block, i, j);

          // Copy back data to global memory.
          graph[(round * block_length + i) * nodes +
                (round * block_length + j)] = block[i][j];
          item.barrier(access::fence_space::local_space);
        });
  });

  q.wait();
}

// Phase 2 of blocked Floyd Warshall algorithm. It always operates on blocks
// that are either on the same row or on the same column of a diagonal block.
void BlockedFloydWarshallPhase2(queue &q, int *graph, int round) {
  // Each group will process one block.
  constexpr auto blocks = block_count;
  // Each item/thread in a group will handle one cell of the block.
  constexpr auto block_size = block_length * block_length;

  q.submit([&](handler &h) {
    LocalBlock diagonal(range<2>(block_length, block_length), h);
    LocalBlock off_diag(range<2>(block_length, block_length), h);

    h.parallel_for<class KernelPhase2>(
        nd_range<1>(blocks * block_size, block_size), [=](nd_item<1> item) {
          auto gid = item.get_group(0);
          auto index = gid;
          if (index != round) {
            auto tid = item.get_local_id(0);
            auto i = tid / block_length;
            auto j = tid % block_length;

            // Copy data to local memory.
            diagonal[i][j] = graph[(round * block_length + i) * nodes +
                                   (round * block_length + j)];
            off_diag[i][j] = graph[(index * block_length + i) * nodes +
                                   (round * block_length + j)];
            item.barrier(access::fence_space::local_space);

            // Compute for blocks above and below the diagonal block.
            BlockedFloydWarshallCompute(item, off_diag, off_diag, diagonal, i,
                                        j);
            // Copy back data to global memory.
            graph[(index * block_length + i) * nodes +
                  (round * block_length + j)] = off_diag[i][j];
```

```
            // Copy data to local memory.
            off_diag[i][j] = graph[(round * block_length + i) * nodes +
                                   (index * block_length + j)];
            item.barrier(access::fence_space::local_space);
            // Compute for blocks at left and at right of the diagonal block.
            BlockedFloydWarshallCompute(item, off_diag, diagonal, off_diag, i,
                                        j);
            // Copy back data to global memory.
            graph[(round * block_length + i) * nodes +
                  (index * block_length + j)] = off_diag[i][j];
            item.barrier(access::fence_space::local_space);
          }
        });
  });
  q.wait();
}

// Phase 3 of blocked Floyd Warshall algorithm. It operates on all blocks
except
// the ones that are handled in phase 1 and in phase 2 of the algorithm.
void BlockedFloydWarshallPhase3(queue &q, int *graph, int round) {
  // Each group will process one block.
  constexpr auto blocks = block_count * block_count;
  // Each item/thread in a group will handle one cell of the block.
  constexpr auto block_size = block_length * block_length;

  q.submit([&](handler &h) {
    LocalBlock A(range<2>(block_length, block_length), h);
    LocalBlock B(range<2>(block_length, block_length), h);
    LocalBlock C(range<2>(block_length, block_length), h);

    h.parallel_for<class KernelPhase3>(
        nd_range<1>(blocks * block_size, block_size), [=](nd_item<1> item) {
          auto bk = round;

          auto gid = item.get_group(0);
          auto bi = gid / block_count;
          auto bj = gid % block_count;

          if ((bi != bk) && (bj != bk)) {
            auto tid = item.get_local_id(0);
            auto i = tid / block_length;
            auto j = tid % block_length;

            // Copy data to local memory.
            A[i][j] = graph[(bi * block_length + i) * nodes +
                            (bk * block_length + j)];
            B[i][j] = graph[(bk * block_length + i) * nodes +
```

```
                            (bj * block_length + j)];
            C[i][j] = graph[(bi * block_length + i) * nodes +
                            (bj * block_length + j)];

            item.barrier(access::fence_space::local_space);

            // Compute.
            BlockedFloydWarshallCompute(item, C, A, B, i, j);

            // Copy back data to global memory.
            graph[(bi * block_length + i) * nodes + (bj * block_length + j)] =
                C[i][j];
            item.barrier(access::fence_space::local_space);
          }
        });
  });

  q.wait();
}

void BlockedFloydWarshall(queue &q, int *graph) {
  for (int round = 0; round < block_count; round++) {
    BlockedFloydWarshallPhase1(q, graph, round);
    BlockedFloydWarshallPhase2(q, graph, round);
    BlockedFloydWarshallPhase3(q, graph, round);
  }
}
```

2.6. Resultados de rendimiento

A continuación, se presentan los resultados obtenidos al realizar pruebas de rendimiento a los algoritmos de Matriz de distancia Euclidiana y al algoritmo de All Pairs Shortest Paths de Floyd Warshall. Los siguientes datos son resultado de las pruebas de rendimiento de los algoritmos antes mencionados de forma tradicional(secuencial) y en forma heterogénea(paralela).

Los tiempos de ejecución presentados a continuación, son el resultado de calcular el promedio de los tiempos de ejecución de cada uno de los algoritmos en su correspondiente escenario. Además, toda la información en crudo recopilada durante las distintas pruebas de comportamiento se puede consultar en el Anexo B.

Para realizar las mediciones de los resultados del algoritmo de la Distancia Euclídea, se utilizaron 6 conjuntos de datos de diferentes tamaños; en todos los casos contienen cuatro columnas de datos y el total de filas depende del conjunto de datos, como se indica en la Tabla 3.

Tabla 3

Datasets Algoritmo de MDE

DataSet	Nro. Filas
DataSet 1	700
DataSet 2	3500
DataSet 3	7000
DataSet 4	10500
DataSet 5	14000
DataSet 6	20000

Nota: Elaboración propia

Para el segundo algoritmo “All Pairs Shortest Paths”, se definió la variable *nodes*, que representa la cantidad de nodos que tiene el grafo en las cuales se va a buscar la ruta más corta de todos los pares de nodos como se muestra en la Tabla 4. El grafo está representado en un arreglo unidimensional que almacena una matriz de adyacencia de un grafo. El tamaño de la matriz es *nodes x nodes.*

Tabla 4

Grafos del All Pairs Shortest Paths

Grafo	Nro. Nodos
Grafo 1	128
Grafo 2	256
Grafo 3	512
Grafo 4	1024
Grafo 5	2048
Grafo 6	4096

Nota: Elaboración propia

El entorno en el que se desarrollaron las pruebas de rendimiento fue sobre el HPC de Intel, la cual es un clúster en el que cuenta con múltiples arquitecturas para computación acelerada como: CPU, GPU y FPGA. El entorno de desarrollo de Intel DevCloud cuenta con el sistema operativo Ubuntu 18.04.3 LTS.

Las CPUs utilizadas para las pruebas fueron:

- Intel(R) Core(TM) i9-11900KB @ 3.30GHz
- Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
- Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz

Las GPUs en la que se realizaron las pruebas son:

- Intel(R) Xeon(R) E-2146G CPU @ 3.50GHz(Intel(R) UHD Graphics P630)
- Intel(R) Core(TM) i9-11900KB @ 3.30GHz(Intel® Iris® Xe MAX)

Para ejecutar las pruebas de rendimiento en dispositivos FPGA se utilizó:

- Intel(R) FPGA Emulation Device

Adicionalmente, se realizó pruebas de rendimiento sobre un ordenador personal con un sistema operativo Windows 11, con el lenguaje de programación C++ que cuenta con las siguientes características: AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz y 16 GB de memoria RAM.

2.6.1. Resultados de rendimiento del algoritmo de MDE en forma secuencial

Los resultados presentados en las siguientes tablas son el promedio de tiempo de procesamiento en segundos al ejecutar el algoritmo de alto coste computacional de Matriz de Distancia Euclidiana de forma secuencial en distintos tipos de procesadores y utilizando 6 datasets de diferente tamaño para monitorear el comportamiento del algoritmo. Las pruebas se ejecutaron en el HPC de Intel y también en una PC personal para tener una comparativa con un entorno de desarrollo tradicional.

En la Tabla 5, se muestra los tiempos de ejecución obtenidos del HPC de Intel en el procesador Intel(R) Xeon(R) E-2146G.

Tabla 5

Tiempos de ejecución de MDE secuencial. Intel(R) Xeon(R) E-2146G CPU @ 3.50GHz

DataSet	Nro. Filas	Tiempo promedio de ejecución (segundos)
DataSet 1	700	0.0468
DataSet 2	3500	1.1488
DataSet 3	7000	4.6414
DataSet 4	10500	10.2781
DataSet 5	14000	18.4560
DataSet 6	20000	34.7430

Nota: Elaboración propia

En la Tabla 6 se muestran los resultados obtenidos al ejecutar el algoritmo de MDE de forma secuencial en el HPC de Intel en el procesador 11th Gen Intel(R) Core(TM) i9-11900KB.

Tabla 6

Tiempos de ejecución de MDE secuencial. 11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz

DataSet	Nro. Filas	Tiempo promedio de ejecución (s)
DataSet 1	700	0.033
DataSet 2	3500	0.8253
DataSet 3	7000	3.3329
DataSet 4	10500	7.2721
DataSet 5	14000	13.2535
DataSet 6	20000	24.4938

Nota: Elaboración propia

En la Tabla 7 se muestra los tiempos de ejecución obtenidos al ejecutar el algoritmo en una PC personal que cuento con el procesador AMD Ryzen 7 5800H.

Tabla 7

Tiempos de ejecución de MDE secuencial. AMD Ryzen 7 5800H - 3.2 GHz

DataSet	Nro. Filas	Tiempo promedio de ejecución (s)
DataSet 1	700	0.0342
DataSet 2	3500	0.9229
DataSet 3	7000	3.6172
DataSet 4	10500	8.0282
DataSet 5	14000	14.0850
DataSet 6	20000	26.6228

Nota: Elaboración propia

2.6.2. Resultados de rendimiento del algoritmo de MDE en forma paralela

A continuación, se presentan los tiempos de ejecución obtenidos al realizar pruebas de rendimiento al algoritmo de Matriz de Distancia Euclidiana en forma paralela utilizando programación heterogénea con 6 datasets de diferente tamaño para monitorear el comportamiento del algoritmo. Para ello, se utilizaron diferentes tipos de aceleradores heterogéneos que se encuentran disponibles en el clúster de HPC de Intel.

Pruebas de rendimiento en CPU del algoritmo de MDE en paralelo

En las tablas presentadas a continuación se detalla el desempeño de diversas arquitecturas de CPUs al ejecutar el algoritmo de Matriz de Distancia Euclidiana en forma paralela. Se puede observar que los tiempos de ejecución obtenidos son óptimos en comparación con los resultados obtenidos al realizar las pruebas de rendimiento al mismo algoritmo en forma secuencial.

Los datos presentados en la Tabla 8 muestran una optimización de tiempo significativa con respecto a el tiempo de procesamiento al utilizar una programación tradicional.

Tabla 8

Tiempos de ejecución de MDE en paralelo. Intel(R) Xeon(R) E-2146G CPU @ 3.50GHz

DataSet	Nro. Filas	Tiempo promedio de ejecución (s)
DataSet 1	700	0.0088
DataSet 2	3500	0.0225
DataSet 3	7000	0.0630
DataSet 4	10500	0.1290
DataSet 5	14000	0.1821
DataSet 6	20000	0.3577

Nota: Elaboración propia

La Tabla 9 contiene los tiempos de procesamiento obtenidos al ejecutar el algoritmo de MDE utilizando programación heterogénea en una CPU Intel Xeon Gold 6128.

Tabla 9

Tiempos de ejecución de MDE en paralelo. Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz

DataSet	Nro. Filas	Tiempo promedio de ejecución (s)
DataSet 1	700	0.0088
DataSet 2	3500	0.0152
DataSet 3	7000	0.0394
DataSet 4	10500	0.0751
DataSet 5	14000	0.0730
DataSet 6	20000	0.1937

Nota: Elaboración propia

De igual manera, la Tabla 10 contiene los tiempos de procesamiento obtenidos al ejecutar el algoritmo de MDE utilizando programación heterogénea en una CPU Intel Core i9-11900KB.

Tabla 10

Tiempos de ejecución de MDE en paralelo. 11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz

DataSet	Nro. Filas	Tiempo promedio de ejecución (s)
DataSet 1	700	0.0051
DataSet 2	3500	0.0137
DataSet 3	7000	0.0458
DataSet 4	10500	0.1029
DataSet 5	14000	0.1760
DataSet 6	20000	0.3491

Nota: Elaboración propia

Los datos recopilados muestran que la CPU Intel Core i9-11900KB es la óptima al ejecutar el algoritmo de MDE de forma paralela, este escenario se repite en los *Datasets 1,2,3,4 y 5*. Para el caso del *Dataset 6,* la CPU Intel Xeon Gold 6128 obtuvo una ligera ventaja frente el Core i9.

Pruebas de rendimiento en GPU del algoritmo de MDE en paralelo

La Tabla 11 contiene los tiempos de ejecución obtenidos al ejecutar el algoritmo de MDE en forma paralela en una GPU. Para ello se escogió la gráfica integrada del procesador Intel Xeon E-2146G, la cual contiene la GPU Intel UHD Graphics P630. Los tiempos de procesamiento obtenidos al realizar las pruebas en un dispositivo gráfico son ligeramente inferiores a los obtenidos el utilizar una CPU, como se muestra a continuación.

Tabla 11

Tiempos de ejecución de MDE en paralelo. Intel UHD Graphics P630

DataSet	Nro. Filas	Tiempo promedio de ejecución (s)
DataSet 1	700	0.0084

DataSet 2	3500	0.0576
DataSet 3	7000	0.1939
DataSet 4	10500	0.4209
DataSet 5	14000	0.6839
DataSet 6	20000	1.3821

Nota: Elaboración propia

Pruebas de rendimiento en FPGA del algoritmo de MDE en paralelo

Los resultados obtenidos de las pruebas de rendimiento en FPGA son emulados (Tabla 12), para ello se utilizó como dispositivo el Intel(R) FPGA Emulation Device. Los tiempos de procesamiento al ejecutar un algoritmo en una FPGA emulada varían a los tiempos de ejecución obtenidos en un hardware FPGA.

Tabla 12

Tiempos de ejecución de MDE en paralelo. Intel(R) FPGA Emulation Device

DataSet	Nro. Filas	Tiempo promedio de ejecución (s)
DataSet 1	700	0.0132
DataSet 2	3500	0.0186
DataSet 3	7000	0.0367
DataSet 4	10500	0.0772
DataSet 5	14000	0.1214
DataSet 6	20000	0.2109

Nota: Elaboración propia

2.6.3. Resultados de rendimiento del algoritmo All Pairs Shortest Paths en forma secuencial

Los resultados presentados en las siguientes tablas muestran el tiempo promedio de procesamiento, medido en segundos, al ejecutar secuencialmente el algoritmo de alto costo computacional All Pairs Shortest Paths(APSP) en diversos tipos de procesadores. Las pruebas se realizaron tanto en el sistema de computación de alto rendimiento (HPC) de Intel como en una PC personal, proporcionando así una comparación con un entorno de desarrollo tradicional.

Los datos de la Tabla 13 indican los tiempos de ejecución promedios obtenidos al ejecutar el algoritmo de All Pairs Shortest Paths en forma secuencial. El procesador utilizado fue la CPU Intel Xeon E-2146G.

Tabla 13

Tiempos de ejecución de APSP secuencial. Intel(R) Xeon(R) E-2146G CPU @ 3.50GHz

Grafo	Nro. Nodos	Tiempo promedio de ejecución (s)
Grafo_1	128	0.0017
Grafo_2	256	0.0104
Grafo_3	512	0.0702
Grafo_4	1024	0.4958
Grafo_5	2048	4.4130
Grafo_6	4096	34.1056

Nota: Elaboración propia

El procesador Intel Core i9-11900KB con el algoritmo de APSP en serial obtuvo mejores tiempos de ejecución en comparación con el procesador Intel Xeon E-2146G como se puede apreciar en la Tabla 14.

Tabla 14

Tiempos de ejecución de APSP secuencial. 11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz

Grafo	Nro. Nodos	Tiempo promedio de ejecución (s)
Grafo_1	128	0.0013
Grafo_2	256	0.0072
Grafo_3	512	0.0470
Grafo_4	1024	0.3325
Grafo_5	2048	2.4640
Grafo_6	4096	24.4084

Nota: Elaboración propia

Para ejecutar el algoritmo de APSP en una computadora personal, fue necesario realizar una adaptación al código debido a la existencia de una restricción al momento de ejecutar el algoritmo de la Figura 40 en el lenguaje C+. El problema se debe a que el número de nodos del *Grafo_4* excedía la capacidad de almacenamiento de la variable

array[], por lo cual se la remplazó por la variable *vector*<> la cual tiene la capacidad de almacenar mayor número de *n* variables.

La Tabla 15 muestra el promedio de tiempo al ejecutar el algoritmo en un procesador AMD Ryzen 7 5800H.

Tabla 15

Tiempos de ejecución de APSP secuencial. AMD Ryzen 7 5800H - 3.2 GHz

Grafo	Nro. Nodos	Tiempo promedio de ejecución (s)
Grafo_1	128	0.0101
Grafo_2	256	0.0833
Grafo_3	512	0.6616
Grafo_4	1024	5.4323
Grafo_5	2048	49.0015
Grafo_6	4096	384.1176

Nota: Elaboración propia

2.6.4. Resultados de rendimiento del algoritmo All Pairs Shortest Paths en forma paralela

A continuación, se presentan los tiempos de ejecución obtenidos al realizar pruebas de rendimiento al algoritmo de All Pairs Shortest Paths utilizando programación heterogénea, aprovechando el procesamiento en paralelo, además se utilizaron 6 datasets para medir el rendimiento del algoritmo con diferentes volúmenes de datos. Para ello, se utilizaron diferentes tipos de aceleradores heterogéneos que se encuentran disponibles en el clúster de HPC de Intel.

Pruebas de rendimiento en CPU del algoritmo All Pairs Shortest Paths en paralelo

En las tablas presentadas a continuación se detalla el desempeño de diversas arquitecturas de CPUs al ejecutar el algoritmo de All Pairs Shortest Paths utilizando la técnica de bloques paralelos. Se puede observar que los tiempos de ejecución obtenidos son óptimos en comparación con los resultados obtenidos al realizar las pruebas de rendimiento al mismo algoritmo en forma secuencial.

Al ejecutar el algoritmo de All Pairs Shortest Paths en el procesador Intel Xeon E-2146G en bloques paralelos, los resultados indican que el tiempo de ejecución no es óptimo, como nos muestra la Tabla 16.

Tabla 16

Tiempos de ejecución de APSP en paralelo. Intel(R) Xeon(R) E-2146G CPU @ 3.50GHz

Grafo	Nro. Nodos	Tiempo promedio de ejecución (s)
Grafo_1	128	0.0027
Grafo_2	256	0.0132
Grafo_3	512	0.0858
Grafo_4	1024	0.6490
Grafo_5	2048	5.1412
Grafo_6	4096	40.7891

Nota: Elaboración propia

La Tabla 17 nos muestra los tiempos de ejecución obtenidos el algoritmo de APSP en bloques paralelos. Intel Xeon Gold 6128 es la CPU que obtuvo el mejor rendimiento el ejecutar el algoritmo en forma paralela.

Tabla 17

Tiempos de ejecución de APSP en paralelo. Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz

Grafo	Nro. Nodos	Tiempo promedio de ejecución (s)
Grafo_1	128	0.0019
Grafo_2	256	0.0053
Grafo_3	512	0.0253
Grafo_4	1024	0.1197
Grafo_5	2048	0.8516
Grafo_6	4096	6.4823

Nota: Elaboración propia

La CPU Intel Core i9-11900KB también redujo el tiempo de ejecución al utilizar la técnica por bloques paralelos con respecto a la ejecución de forma serial como se muestra en la Tabla 18.

Tabla 18

Tiempos de ejecución de APSP en paralelo. 11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz

Grafo	Nro. Nodos	Tiempo promedio de ejecución (s)
Grafo_1	128	0.0011
Grafo_2	256	0.0051
Grafo_3	512	0.0315
Grafo_4	1024	0.2666
Grafo_5	2048	2.1082
Grafo_6	4096	16.7158

Nota: Elaboración propia

Pruebas de rendimiento en GPU del algoritmo All Pairs Shortest Paths en paralelo

A continuación, se muestran las pruebas de rendimiento del algoritmo de All Pairs Shortest Paths en bloques paralelos utilizando unidades de procesamiento gráfico.

En la Tabla 19 se muestra los resultados de rendimiento de la GPU integrada del procesador Intel Xeon E-2146G (Intel UHD Graphics P630).

Tabla 19

Tiempos de ejecución de APSP en paralelo. Intel(R) UHD Graphics P630

Grafo	Nro. Nodos	Tiempo promedio de ejecución (s)
Grafo_1	128	0.0016
Grafo_2	256	0.0047
Grafo_3	512	0.0195
Grafo_4	1024	0.1151
Grafo_5	2048	0.8446
Grafo_6	4096	6.4934

Nota: Elaboración propia

Los datos de la Tabla 20 nos muestra los tiempos de ejecución obtenidos al ejecutar el algoritmo de All Pairs Shortest Paths en bloques paralelos utilizando la GPU Intel UHD Graphics del procesador Corei9 de 11va generación.

Tabla 20

Tiempos de ejecución de APSP en paralelo. Intel(R) UHD Graphics

Grafo	Nro. Nodos	Tiempo promedio de ejecución (s)

Grafo_1	128	0.0009
Grafo_2	256	0.0024
Grafo_3	512	0.0098
Grafo_4	1024	0.0575
Grafo_5	2048	0.4097
Grafo_6	4096	3.1755

Nota: Elaboración propia

Pruebas de rendimiento en FPGA del algoritmo All Pairs Shortest Paths en paralelo

Los resultados obtenidos de las pruebas de rendimiento en FPGA son emulados (Tabla 21), para ello se utilizó como dispositivo el Intel(R) FPGA Emulation Device. Los tiempos de procesamiento al ejecutar un algoritmo en una FPGA emulada varían a los tiempos de ejecución obtenidos en un hardware FPGA.

Tabla 21

Tiempos de ejecución de APSP en paralelo. Intel(R) FPGA Emulation Device

Grafo	Nro. Nodos	Tiempo promedio de ejecución (s)
Grafo_1	128	0.0016
Grafo_2	256	0.0056
Grafo_3	512	0.0193
Grafo_4	1024	0.1131
Grafo_5	2048	0.8385
Grafo_6	4096	6.7233

Nota: Elaboración propia

CAPÍTULO 3

Validación y resultados

3.1. Métricas de evaluación

Para llevar a cabo la evaluación y análisis de los resultados obtenidos, se dispone de diversos criterios que permiten valorar la calidad y rendimiento de los algoritmos (Naiouf et al., 2021).

Existen varias métricas para evaluar el rendimiento de un algoritmo paralelo, en las cueles se encuentran las más comunes: tiempo de ejecución, SpeedUp y la eficiencia. Por otra parte, es importante evaluar la escalabilidad la cual permite capturar las características de un algoritmo paralelo y la arquitectura en que se lo implementa (Naiouf et al., 2021).

3.1.1. Tiempo de ejecución

El tiempo de ejecución es una medida que hace referencia al tiempo que se tarda un algoritmo en completar una instrucción. El tiempo de procesamiento es un indicativo eficaz de la eficiencia de un programa, ya sea tanto serial como paralelo. Esta métrica se realiza contando el tiempo transcurrido desde el inicio hasta el final de la ejecución de un algoritmo.

Es importante considerar que el tiempo de ejecución de un algoritmo, a pesar de que tenga la misma cantidad de datos y se encuentre en el mismo entorno de ejecución, existen ligeras variaciones de tiempo de procesamiento, para ello, es necesario realizar varias mediciones de tiempo para sacar el promedio de tiempo de ejecución.

3.1.2. SpeedUp

El SpeedUp o también conocido como aceleración es una métrica frecuentemente utilizada para comparar el rendimiento entre un algoritmo implementado en paralelo y un algoritmo en implementado en secuencial.

Para medir la efectividad de un programa paralelizado frente a la versión secuencial, se debe medir los tiempos de procesamiento de cada uno y con ellos calcular la relación entre los tiempos de ejecución o Speedup:

- ***Tsec*** = Tiempo de ejecución del programa secuencial
- ***Tpar*** = Tiempo de ejecución del programa en paralelo

$$Sp = \frac{Tsec}{Tpar}$$

El SpeedUp se interpreta en base a 1, es decir, cuanto mayor sea el resultado con respecto a 1, mejor es el rendimiento del algoritmo en forma paralela. Si en resultado es iguala 1, significa que existe ninguna mejora de rendimiento al paralelizar el algoritmo, el tiempo de ejecución es el mismo en ambas versiones, tanto secuencial como en paralelo. En el caso de que el resultado sea menor a 1, significa que la versión paralela es más lenta que la versión secuencial, pierde rendimiento.

3.2. Comparativa de tiempo de procesamiento

3.2.1. Comparativa de tiempo de procesamiento del algoritmo de MDE

Para realizar la comparativa de tiempo de procesamiento del algoritmo de MDE, se escogió el mejor tiempo de ejecución obtenido en la implementación en paralelo en diferentes arquitecturas.

Para la comparación de tiempos de ejecución, se seleccionaron los dispositivos de hardware que mostraron los mejores resultados en términos de tiempos de procesamiento:

- **Secuencial PC – personal**: AMD Ryzen 7 5800H - 3.2 GHz
- **Secuencial HPC**: 11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz
- **Paralelo CPU**: Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
- **Paralelo GPU**: Intel(R) Xeon(R) E-2146G CPU @ 3.50GHz - Intel(R) UHD Graphics P630
- **Paralelo FPGA**: Intel(R) FPGA Emulation Device

En la Figura 42 se puede observar una comparativa a detalle del algoritmo de la Matriz de Distancia Euclidiana utilizando el datase_1, en donde se puede apreciar una clara ventaja de rendimiento de algoritmo implementado en forma paralela frente a la implementación en secuencial.

Al medir la aceleración utilizando el factor speedup, tomando en cuenta la implementación en paralelo con mejor tiempo, que en este caso fue la GPU frente a la implementación secuencial en una PC personal, obtenemos que la versión paralela en GPU es *4,071x* veces más rápida que la versión secuencial. La implementación en GPU también nos indica que es *3,929x* veces más rápido que la versión secuencial ejecutada en HPC.

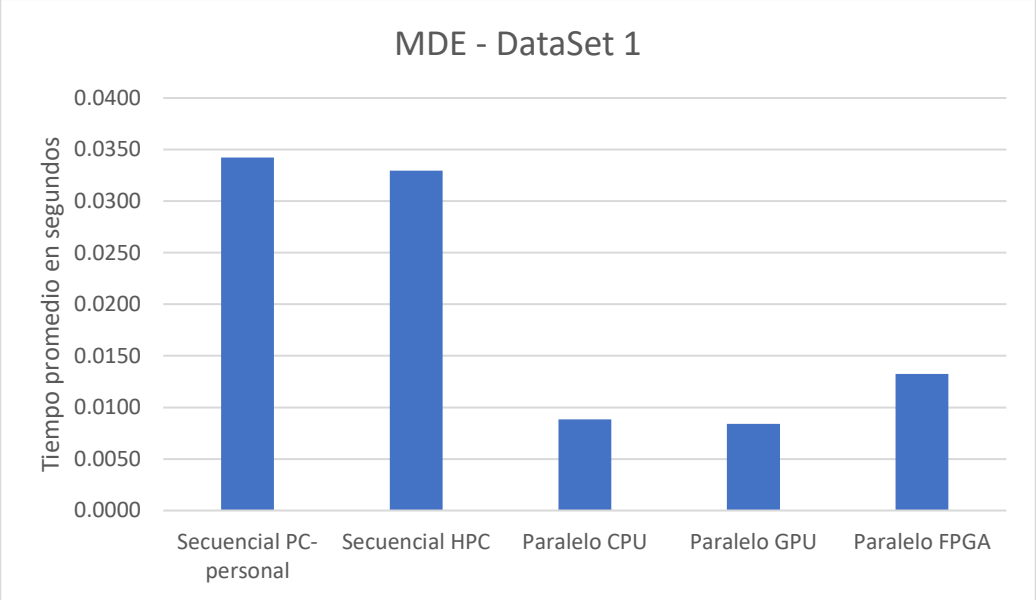


Figura 42. Comparativa de tiempos de procesamiento del algoritmo MDE (dataset_1)

Fuente: Elaboración propia

Los tiempos de ejecución utilizando el dataset_2 muestran que la implementación pelarela en CPU tiene el mejor rendimiento de tiempo. El algoritmo paralelo en CPU es *60,703x* veces más rápido que la versión secuencial en una PC personal y *54.285x* veces más rápida que la versión secuencial en HPC.

En la Figura 43 nos muestra también una mejora en la implementación paralela en CPU con respecto a ejecución en GPU, es decir, que en la CPU se ejecuta 3.788x veces más rápido que en la GPU.

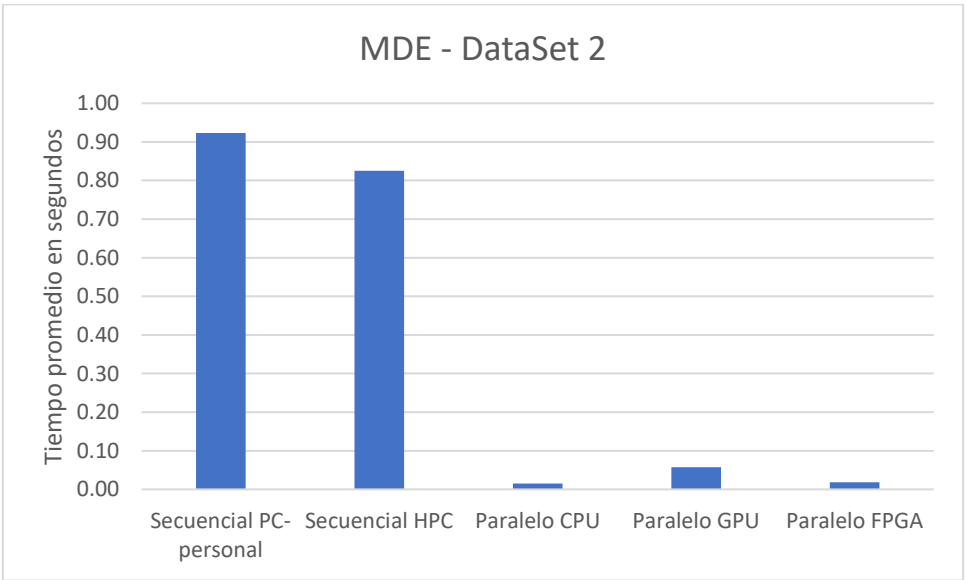


Figura 43. Comparativa de tiempos de procesamiento del algoritmo MDE (dataset_2)

Fuente: Elaboración propia

La Figura 44 muestra una aceleración de la versión paralela en CPU de *91.922x* veces más rápido que el algoritmo ejecutado secuencialmente en un PC personal, además indica que es *84.697x* veces más rápido que la ejecución secuencial en HPC.

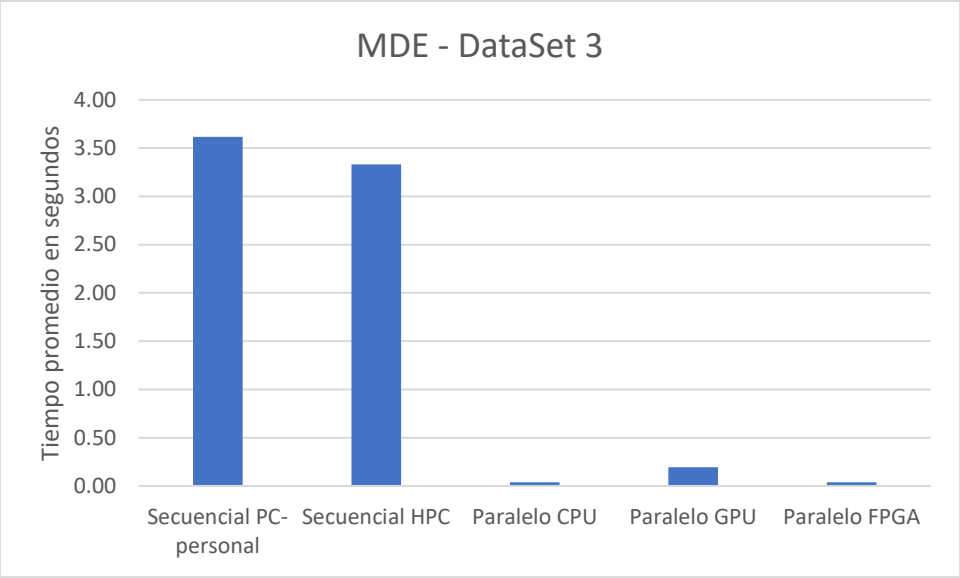


Figura 44. Comparativa de tiempos de procesamiento del algoritmo MDE (dataset_3)

Fuente: Elaboración propia

El speedup calculado utilizando el dataset_4 es de *106.918x* entre el tiempo de procesamiento en paralelo con CPU y el tiempo de procesamiento en secuencial con una PC personal como se muestra en la Figura 45.

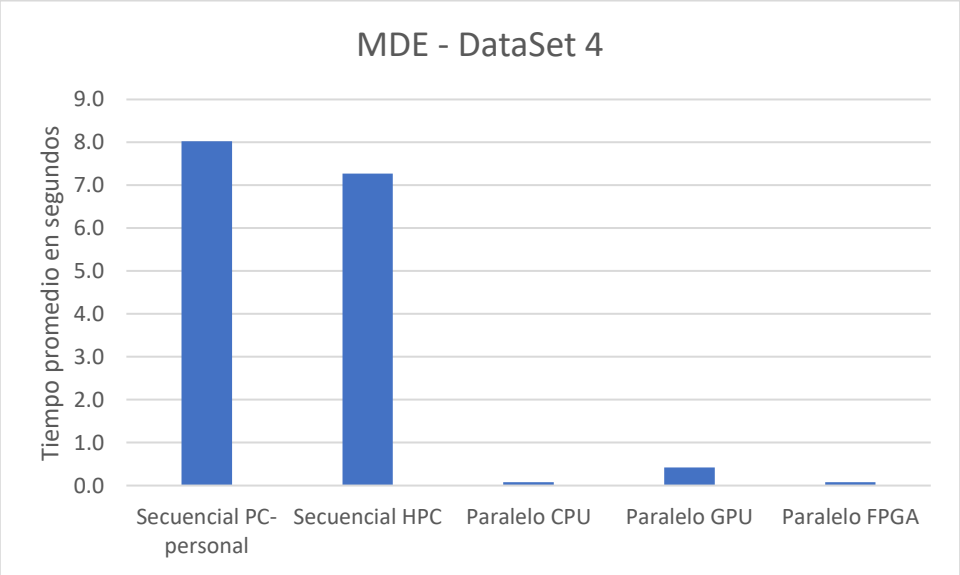


Figura 45. Comparativa de tiempos de procesamiento del algoritmo MDE (dataset_4)

Fuente: Elaboración propia

La Figura 46 indica una mayor diferencia al calcular el speedup que pasa a ser *192.89x* entre la implementación secuencial en una PC personal y la implementación en

paralelo en CPU, es el resultado más alto alcanzado al ejecutar el algoritmo de MDE con todos los datasets. El tiempo de procesamiento paralelo en CPU es *181.503x* veces más rápida que al ejecutar el algoritmo de forma secuencial en HPC.

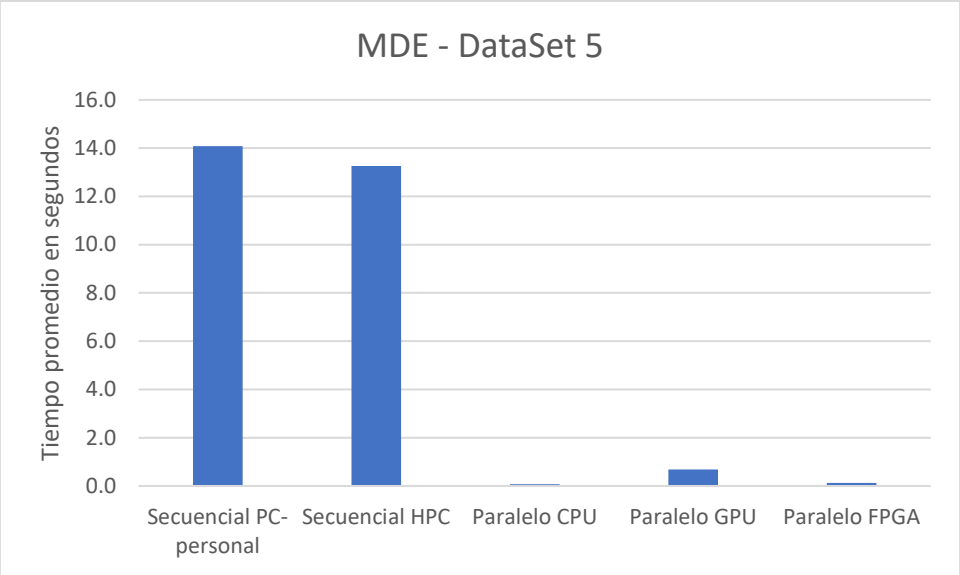


Figura 46. Comparativa de tiempos de procesamiento del algoritmo MDE (dataset_5)

Fuente: Elaboración propia

Al realizar las pruebas de rendimiento con el dataset_6, el speedup calculado es de *137.449x* entre el procesamiento secuencial en una computadora personal y el procesamiento en paralelo en una CPU de Intel. En la Figura 47 también se puede apreciar que la implementación paralela en CPU es 126.457x veces más rápida que al ejecutarla de forma secuencial en HPC.

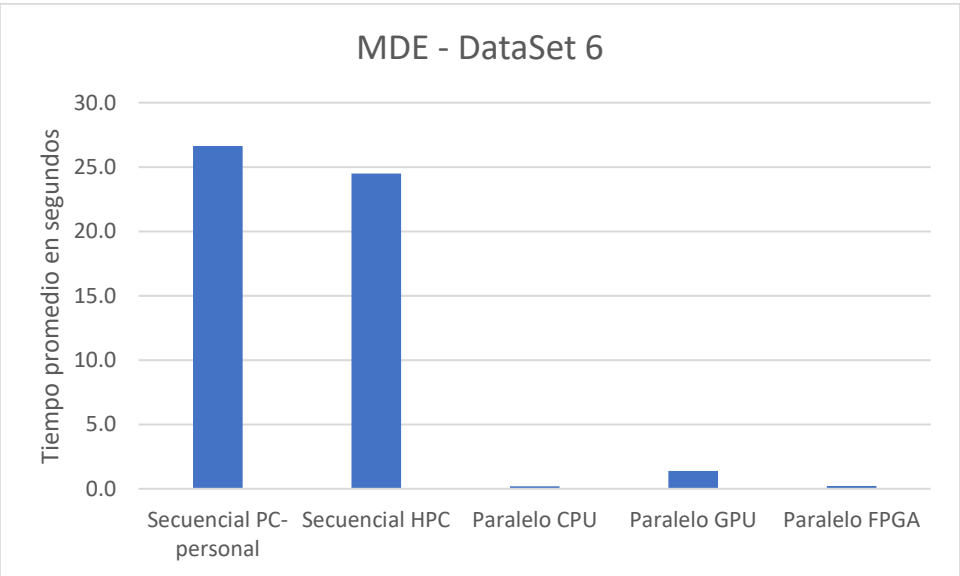


Figura 47. Comparativa de tiempos de procesamiento del algoritmo MDE (dataset_6)

Fuente: Elaboración propia

3.2.2. Comparativa de tiempo de procesamiento del algoritmo de All Pairs Shortest Paths

Para realizar la comparativa de tiempo de procesamiento del algoritmo de APSP, se escogió el mejor tiempo de ejecución obtenido en la implementación en paralelo en diferentes arquitecturas.

Para la comparación de tiempos de ejecución, se seleccionaron los dispositivos de hardware que mostraron los mejores resultados en términos de tiempos de procesamiento:

- **Secuencial PC – personal**: AMD Ryzen 7 5800H - 3.2 GHz
- **Secuencial HPC**: 11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz
- **Paralelo CPU**: Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
- **Paralelo GPU**: 11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz - Intel(R) UHD Graphics
- **Paralelo FPGA**: Intel(R) FPGA Emulation Device

En la Figura 48 se muestra una comparación detalla de algoritmo de la All Pairs Shortest Paths utilizando el grafo_1, en donde se puede apreciar una clara ventaja de rendimiento del algoritmo implementado en forma paralela frente a la implementación en secuencial.

Al medir la aceleración utilizando el factor speedup, tomando en cuenta la implementación en paralelo con mejor tiempo, que en este caso fue la GPU, frente a la implementación secuencial en una PC personal, obtenemos que la versión paralela en GPU es *11.619x* veces más rápida que la versión secuencial. La implementación en GPU también nos indica que es *1.448 x* veces más rápido que la versión secuencial ejecutada en HPC.

En la Figura 48 se puede apreciar que el tiempo de procesamiento en forma secuencial en HPC es más eficiente que el procesamiento paralelo en CPU y en FPGA, específicamente es *1.532x* veces más rápido que la CPU y *1.304x* veces más rápida que la FPGA.

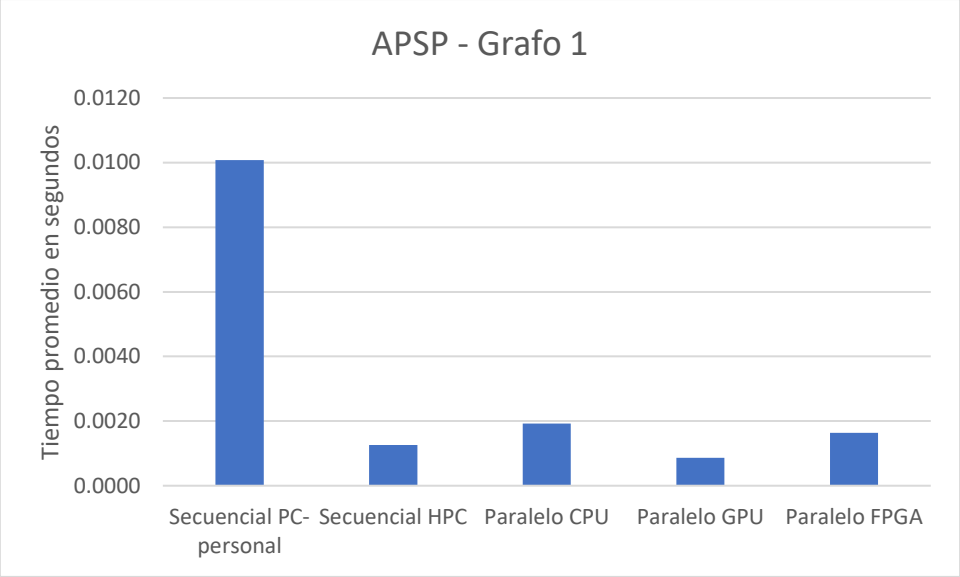


Figura 48. Comparativa de tiempos de procesamiento del algoritmo APSP (grafo_1)

Fuente: Elaboración propia

En las pruebas de rendimiento realizadas con el grafo_2, el tiempo de procesamiento de la CPU y FPGA en paralelo superan al tiempo de procesamiento del algoritmo en secuencial ejecutado en HPC según la Figura 49. La implementación del algoritmo en GPU resultó ser óptimo, siendo *34.029x* veces más rápido que la implementación secuencial en una PC personal y *2.956x* veces más rápido que la implementación secuencial en HPC.

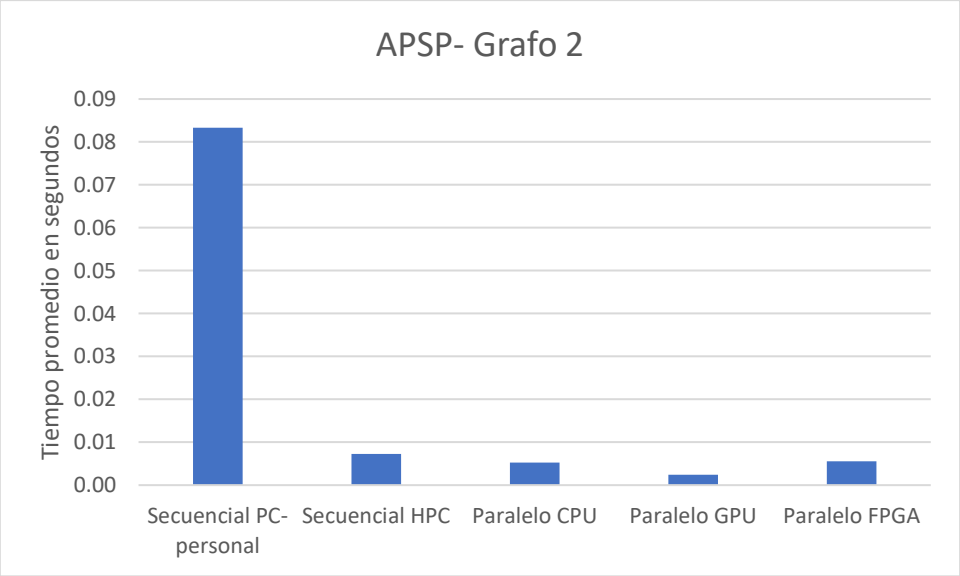


Figura 49. Comparativa de tiempos de procesamiento del algoritmo APSP (grafo_2)

Fuente: Elaboración propia

Las pruebas de rendimiento con el grafo_3 indican que el algoritmo paralelo ejecutado en una GPU obtuvo el mejor promedio tiempo de procesamiento (Figura 50),

siendo *67.374x* veces más rápido que la implementación secuencial en una PC personal y *4.786x* veces más rápido que la implementación secuencial en HPC.

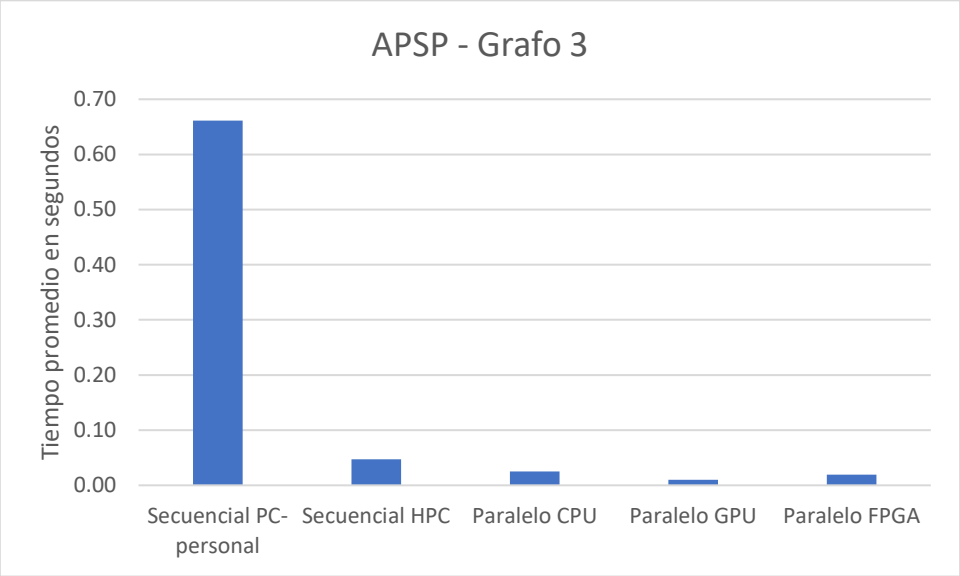


Figura 50. Comparativa de tiempos de procesamiento del algoritmo APSP (grafo_3)

Fuente: Elaboración propia

Las pruebas de rendimiento con el grafo_4 indican que el algoritmo paralelo ejecutado en una GPU obtuvo el mejor promedio tiempo de procesamiento (Figura 51), siendo *94.438x* veces más rápido que la implementación secuencial en una PC personal y *5.780x* veces más rápido que la implementación secuencial en HPC.

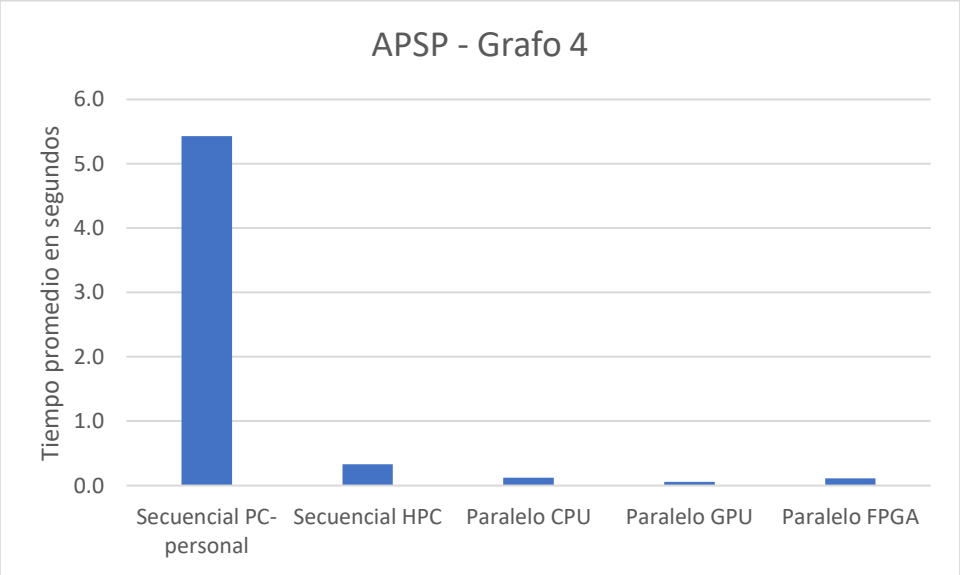


Figura 51. Comparativa de tiempos de procesamiento del algoritmo APSP (grafo_4)

Fuente: Elaboración propia

Como lo muestra la Figura 52, las pruebas de rendimiento con el grafo_5 indican que el algoritmo paralelo ejecutado en una GPU obtuvo el mejor promedio tiempo de

procesamiento, siendo *119.600x* veces más rápido que la implementación secuencial en una PC personal y *6.014x* veces más rápido que la implementación secuencial en HPC.

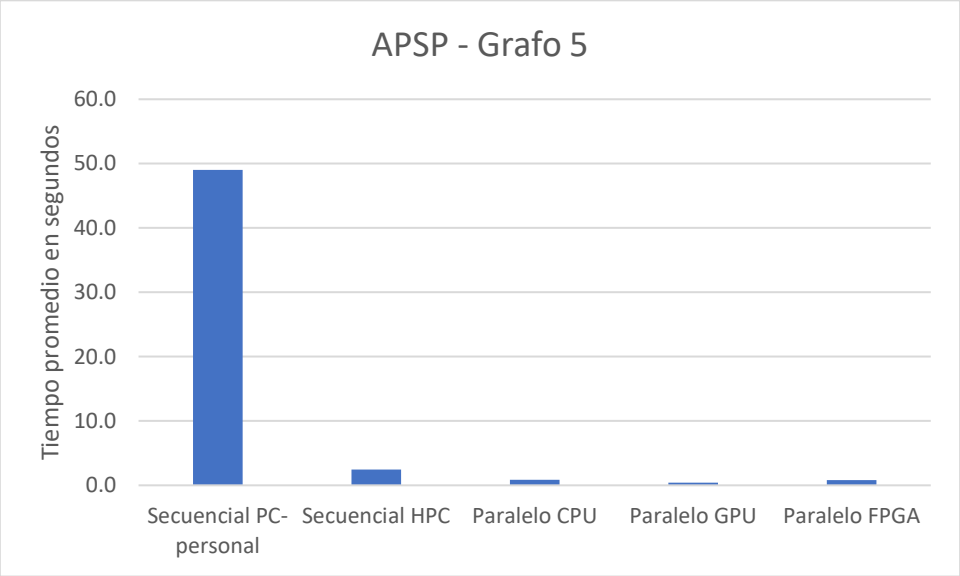


Figura 52. Comparativa de tiempos de procesamiento del algoritmo APSP (grafo_5)

Fuente: Elaboración propia

Como lo muestra la Figura 53, las pruebas de rendimiento con el grafo_6 indican que el algoritmo paralelo ejecutado en una GPU obtuvo el mejor promedio tiempo de procesamiento, siendo *120.961x* veces más rápido que la implementación secuencial en una PC personal y *7.686x* veces más rápido que la implementación secuencial en HPC. La ejecución en paralelo en GPU es *2.041x* veces más rápido que la CPU.

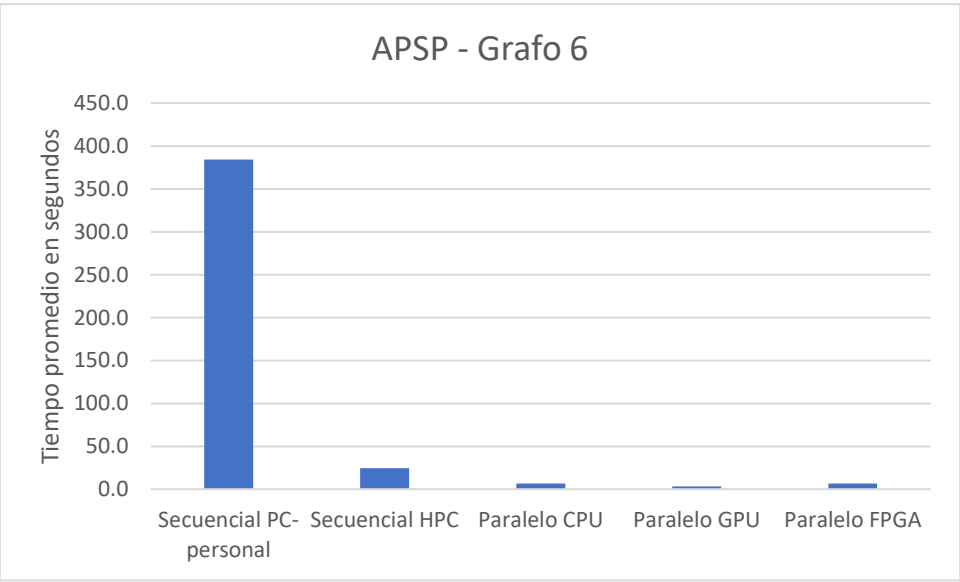


Figura 53. Comparativa de tiempos de procesamiento del algoritmo APSP (grafo_6)

Fuente: Elaboración propia

3.3. Interpretación de resultados

3.3.1. Análisis del tiempo de procesamiento del algoritmo de MDE

El objetivo de realizar pruebas de rendimiento en diferentes arquitecturas es aprovechar la heterogeneidad que nos ofrece el algoritmo de DPC++ y realizar una comparativa de rendimiento entre los diferentes tipos de procesadores que tiene disponible la plataforma de Intel.

La Figura 54 nos muestra la curva de rendimiento de distintos tipos de arquitecturas al ejecutar el algoritmo de Matriz de Distancia Euclidiana con distintos tamaños de dataset (Tabla 3). La gráfica muestra que las implementaciones secuenciales empeoran la eficiencia mientras más grande sea el dataset y su curva de crecimiento es exponencial.

En el algoritmo de Matriz de Distancia Euclidiana la CPU Intel Xeon Gold 6128 mostró ser la más optima, con un promedio de *0.1937s* con el dataset_6 es *137.449x* veces más rápido que la implementación secuencial en una PC personal con un procesador AMD Ryzen 7 5800H - 3.2 GHz y *126.457x* veces más rápido que el procesamiento secuencial en HPC utilizando la CPU Intel Core i9-11900KB - 3.30GHz. Los tiempos de ejecución de la CPU y la FPGA son muy parecidos, por ese motivo se superponen en la gráfica.

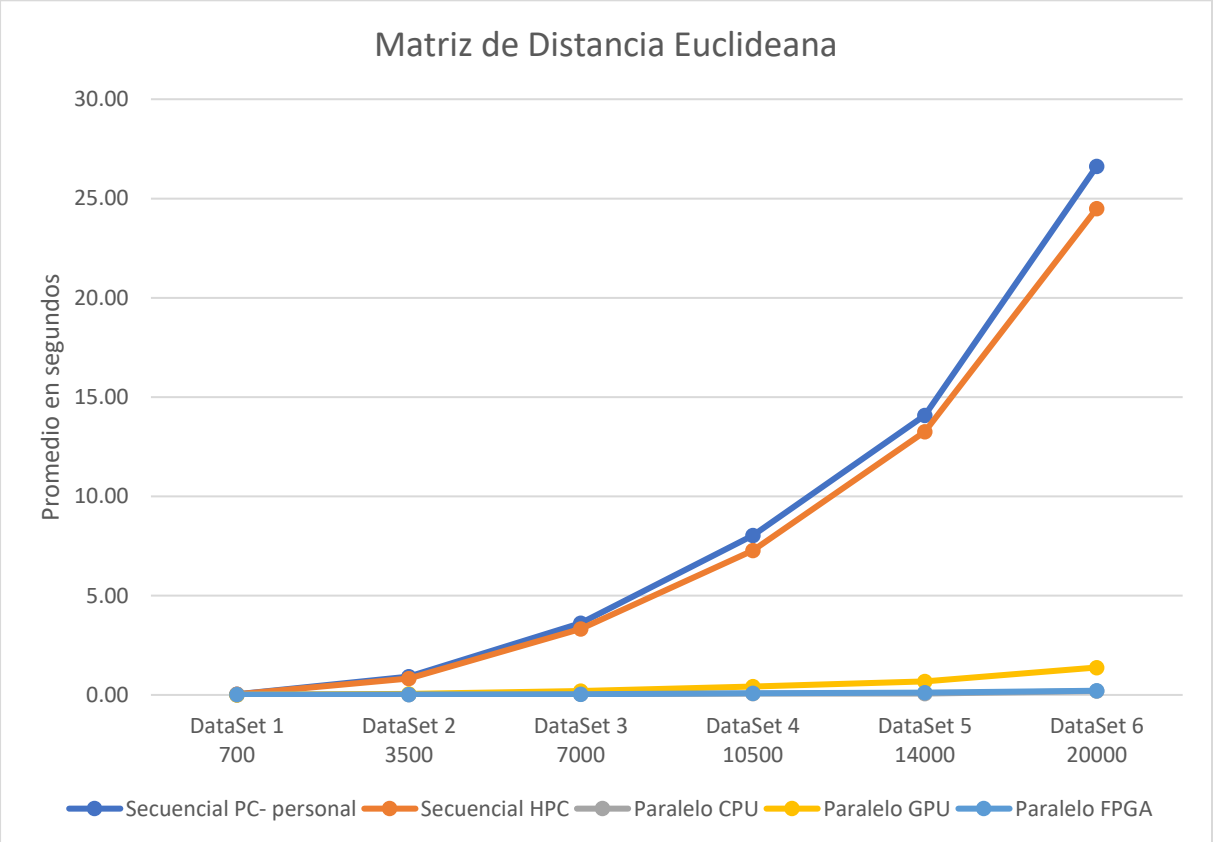


Figura 54. Curva de crecimiento del algoritmo MDE

Fuente: Elaboración propia

3.3.2. Análisis del tiempo de ejecución del algoritmo de All Pairs Shortest Paths

Para el algoritmo de All Pairs Shortest Paths se utilizaron grafos de distinto número de nodos (Tabla 4) para evaluar el tiempo de procesamiento al ejecutar el algoritmo en paralelo y en distintos tipos de arquitecturas.

La Figura 55 muestra que implementación en secuencial en una PC personal es el menor rendimiento presenta ya que obtuvo un promedio de tiempo de procesamiento de *384.1176s* con el grafo_6. La implementación paralela en la GPU integrada del Intel Core i9-11900KB (Intel UHD Graphics) mostró el mejor tiempo de procesamiento con *3.1755s* que es *120.961x* más rápida que la versión secuencial en una PC personal y *7.68x* más rápida que la versión secuencial en HPC. Los tiempos de ejecución de la CPU y la FPGA son muy parecidos, por ese motivo se superponen en la gráfica.

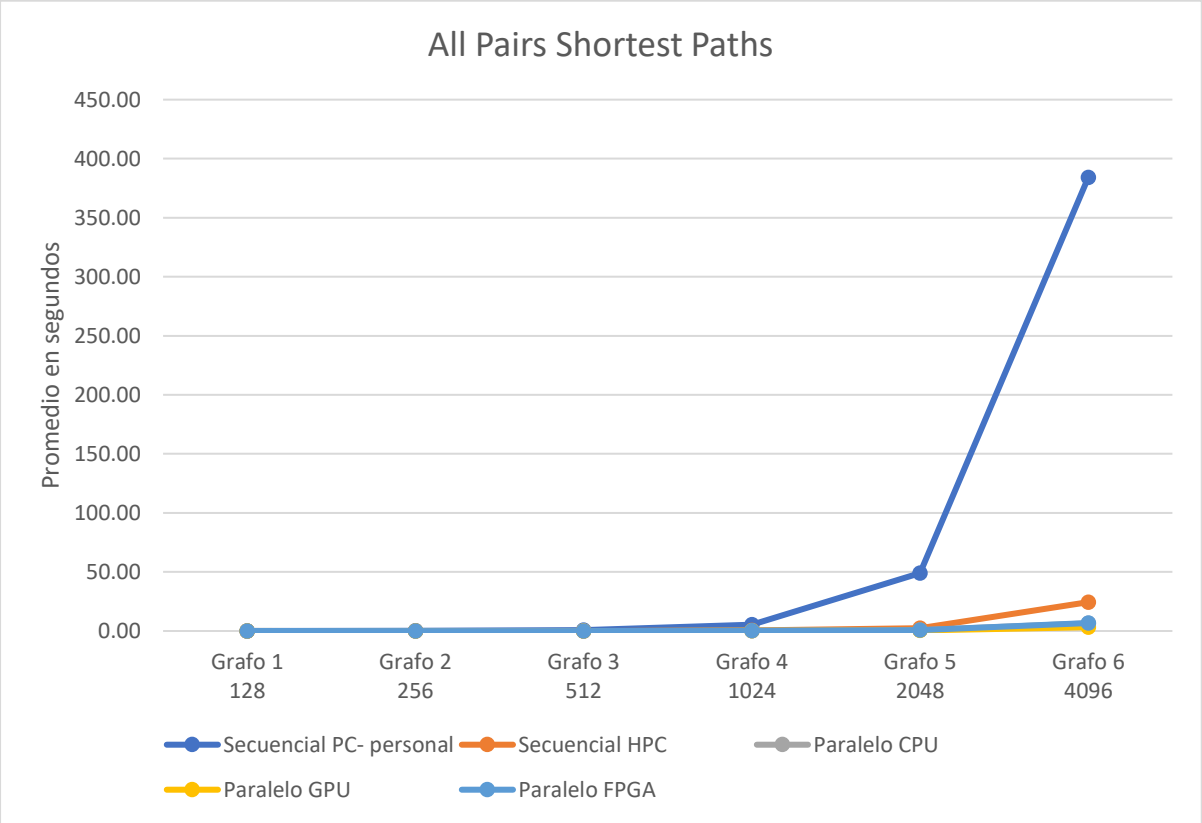


Figura 55. Curva de crecimiento del algoritmo All Pairs Shortest Paths

Fuente: Elaboración propia

Como se puede observar en la Figura 55, que al utilizar los grafos 5 y 6 en una PC personal, los tiempos de ejecución se disparan y se distancian mucho de los tiempos

alcanzados al ejecutarlos en la HPC de Intel. Por ello se decidió realizar pruebas de rendimiento omitiendo la ejecución en secuencial en un PC persona. Para ello de añadió dos grafos con mayor número de nodos: **grafo 7** con 8192 nodos y **grafo 8** con 16384 con 16384 nodos.

En la Tabla 22 se puede observar los resultados de las pruebas de rendimiento al ejecutar el algoritmo APSP en forma secuencial en el procesador CPU Intel Xeon Gold 6128.

Tabla 22

Tiempos de ejecución de APSP en secuencial con grafos adicionales. Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz

Grafo	Nro. Nodos	Tiempo promedio de ejecución (s)
Grafo_7	8192	285.6187
Grafo_8	16384	2526.67

Nota: Elaboración propia

Las pruebas de rendimiento con los dos grafos adicionales también se ejecutaron en forma paralela en la CPU como se detalla en la Tabla 23.

Tabla 23

Tiempos de ejecución de APSP en paralelo con grafos adicionales. Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz

Grafo	Nro. Nodos	Tiempo promedio de ejecución (s)
Grafo_7	8192	137.4724
Grafo_8	16384	848.99

Nota: Elaboración propia

La Tabla 24 muestra los resultados obtenidos al ejecutar el algoritmo de APSP paralelo en la GPU integrada Intel UHD Graphics.

Tabla 24

Tiempos de ejecución de APSP en paralelo con grafos adicionales. Intel(R) UHD Graphics

Grafo	Nro. Nodos	Tiempo promedio de ejecución (s)
Grafo_7	8192	25.0217
Grafo_8	16384	198.91

Nota: Elaboración propia

Los resultados obtenidos de las pruebas de rendimiento en FPGA son emulados (Tabla 25), para ello se utilizó como dispositivo el Intel(R) FPGA Emulation Device. Los

tiempos de procesamiento al ejecutar un algoritmo en una FPGA emulada varían a los tiempos de ejecución obtenidos en un hardware FPGA.

Tabla 25

Tiempos de ejecución de APSP en paralelo con grafos adicionales. Intel(R) FPGA Emulation Device

Grafo	Nro. Nodos	Tiempo promedio de ejecución (s)
Grafo_7	8192	106.1309
Grafo_8	16384	849.88

Nota: Elaboración propia

En la curva de crecimiento de la Figura 56, la implementación paralela en GPU obtuvo el mejor promedio de tiempo de procesamiento, *198.913s* al ejecutar el algoritmo con el grafo 8. La GPU fue *12.7024x* veces más rápida que la implementación secuencial en HPC y *4.2681x* veces más rápido que la implementación paralela en CPU.

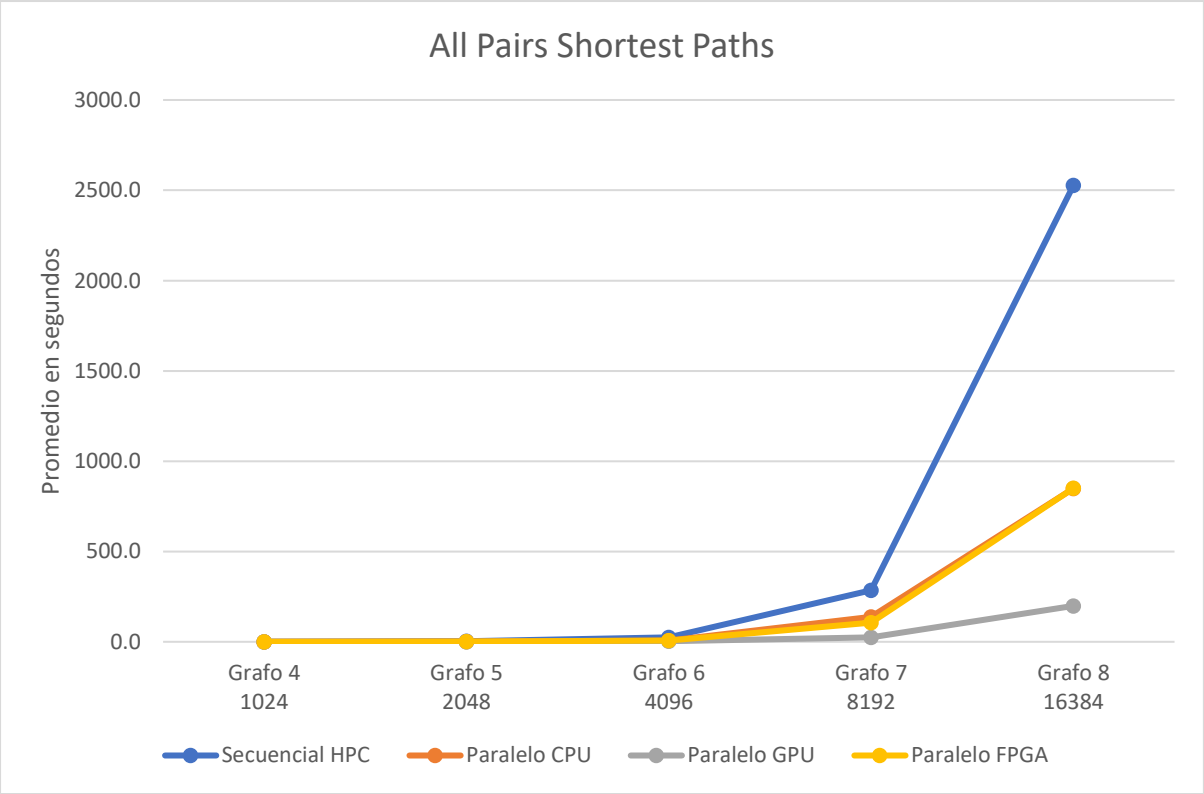


Figura 56. Curva de crecimiento del algoritmo All Pairs Shortest Paths con grafos adicionales

Fuente: Elaboración propia

CONCLUSIONES

La utilización de lenguajes de programación como DPC++ centrados en desarrollo para sistemas heterogéneos, ahorran mucho tiempo al programador ya que, no es necesario aprender diferentes tipos de lenguajes para desarrollar algoritmos en varios tipos de arquitecturas, esto proporciona flexibilidad para aprovechar diferentes recursos según la disponibilidad y el rendimiento requerido.

La plataforma de Intel DevCloud proporciona un amplio catálogo de herramientas bien documentadas para el desarrollo en HPC, IA y programación heterogénea. También cuenta con una comunidad activa donde brindan ayuda para resolver problemas e inquietudes a los usuarios de la plataforma.

C++ permite un control de bajo nivel sobre los recursos del sistema y ofrece un alto rendimiento. Esto lo hace ideal para aplicaciones que requieren eficiencia en términos de velocidad de ejecución y uso de recursos.

Los algoritmos implementados secuencialmente bajan su rendimiento exponencialmente al aumentar el tamaño del problema en comparación al procesamiento en paralelo que mantiene un buen rendimiento incluso al procesar grandes volúmenes de datos.

Los resultados obtenidos en las pruebas de rendimiento han demostrado que, en los distintos casos estudiados, la implementación DPC++ supera a la implementación C++. Esto se ha demostrado mediante la medición de los tiempos de ejecución y el cálculo de la aceleración de los algoritmos. DPC++ ha mostrado ser un instrumento poderoso y eficiente para la optimización de algoritmos de alto coste computacional, ofreciendo evidentes ventajas en términos de rendimiento y utilización de recursos frente a una programación secuencial.

Al analizar las curvas de crecimiento de los algoritmos implementados, se puede concluir que el procesamiento secuencial baja su rendimiento a medida que se aumentan los datos de entrada de forma más agresiva que el procesamiento en paralelo.

En el proceso de desarrollo de la tesis, he experimentado un crecimiento tanto en el ámbito académico como en el personal. Este proyecto ha sido una lección de perseverancia y dedicación debido a la curva de aprendizaje, ya que al principio me costó asimilar los conocimientos necesarios para dominar la programación heterogénea. Sin embargo, gracias a la práctica constante y a las muchas horas dedicadas al estudio, mis esfuerzos finalmente dieron resultados y logré cumplir el objetivo de finalizar el proyecto de tesis.

RECOMENDACIONES

Es recomendable que, al momento de trabajar con matrices, se evite utilizar estructuras de datos como arreglos o vectores en forma bidimensional cuando se intente representar una matriz. Esto se debe a que, al asignar este tipo de estructura de datos a un buffer, pueden existir errores o ser necesario utilizar un bucle para llenar correctamente la matriz. Lo recomendable es representar la matriz en un arreglo unidimensional, como se muestra en los ejemplos desarrollados por Intel.

DPC++ permite a los desarrolladores describir la funcionalidad del hardware utilizando construcciones de programación de alto nivel, lo que puede simplificar el diseño y reducir el tiempo de desarrollo. Si la prioridad al implementar un algoritmo paralelo en una arquitectura específica y tener un mayor control del hardware, es recomendable utilizar lenguajes especializados en programación de bajo nivel como por ejemplo VHDL o Verilog que son lenguajes de descripción de hardware para FPGAs.

A partir de los resultados obtenidos, se ha comprobado que DPC++ mejora notablemente el rendimiento de los algoritmos de alto consumo computacional. Se recomienda que los proyectos que necesiten computación de altas prestaciones consideren la posibilidad de adoptar un lenguaje de estas características. Integrar DPC++ en el proceso de desarrollo y formar a los desarrolladores en su uso eficiente.

Se recomienda desarrollar un trabajo futuro relacionado a la programación de alto rendimiento comparando el lenguaje DPC++ con diferentes lenguajes de bajo nivel como Verilog, CUDA, OpenCL, entre otros.

BIBLIOGRAFÍA

Barney, B., Frederick, D., & Livermore, C. (2018, May 15). *Introduction to Parallel Computing Tutorial*. HPC @ LLNL. https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial

BasuMallick, C. (2022, August 26). *What Is Parallel Processing? Definition, Types, and Examples*. Spiceworks. https://www.spiceworks.com/tech/iot/articles/what-is-parallel-processing/

Bernal, F., Albarracín, C., Gaona, J., Giraldo, L., Mosquera, C., Peña, S., Torres, Y., Ovalle, J., Nieto, J., Chacón, D., Salcedo, S., Suarez, A., Cortés, D., Pinzón, J., Higuera, P., & Baquero, C. (2017, February 5). *Programación Paralela*. http://ferestrepoca.github.io/paradigmas-de-programacion/paralela/paralela_teoria/index.html

Britannica, & T. Editors of Encyclopaedia. (2023, March 10). *Computational complexity*. Encyclopedia Britannica. https://www.britannica.com/topic/computational-complexity

Daroch, D., & Antiñire, S. (2018). *ESTUDIO DEL MUNDO DE LAS FPGAS COMPARATIVAS E IMPLEMENTACION* [UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA]. https://repositorio.usm.cl/handle/11673/46840

Demey, J., Pla, L., Vicente Villardon, J. L., Di Rienzo, J., & Casanoves, F. (2011). *MEDIDAS DE DISTANCIA Y SIMILITUD* (pp. 47–59). https://www.researchgate.net/publication/260137073

Embedded Staff. (2013, February 25). *Introduction to the Multicore Programming Practices Guide*. Embedded. https://www.embedded.com/introduction-to-the-multicore-programming-practices-guide/

Evanvzuk, S. (2013, February 19). *Best practices for multicore programming*. EDN. https://www.edn.com/best-practices-for-multicore-programming/

Feldman, M. (2016, November 8). *Good Times for FPGA Enthusiasts*. Top 500. The List. https://www.top500.org/news/good-times-for-fpga-enthusiasts/

Firesmith, D. (2017). *Multicore Processing*. https://insights.sei.cmu.edu/blog/multicore-processing/

Gizopoulos, D., Papadimitriou, G., Chatzidimitriou, A., Reddi, V. J., Salami, B., Unsal, O. S., Kestelman, A. C., & Leng, J. (2019). Modern Hardware Margins: CPUs, GPUs, FPGAs Recent System-Level Studies. *2019 IEEE 25th International Symposium on On-Line*

Testing and Robust System Design (IOLTS), 129–134. https://doi.org/10.1109/IOLTS.2019.8854386

Graph Everywhere. (2019, June 17). *Algoritmo de distancia euclidiana*. https://www.grapheverywhere.com/algoritmo-de-distancia-euclidiana/

Guamán Rivera, B. L. (2017). *Análisis de rendimiento de un clúster HPC y, arquitecturas manycore y multicore* [Universidad de Cuenca]. http://dspace.ucuenca.edu.ec/ bitstream/123456789/28554/1 /Trabajo%20de%20Titulaci% C3%B3n.pdf

Hagoort, N. (2020, October 24). *Exploring the GPU Architecture*. VMware, Inc. https://core.vmware.com/resource/exploring-gpu-architecture

Hernández Baez, I. (2015). *K-Medias en RapidMiner* [Benemérita Universidad Autónoma de Puebla]. https://repositorioinstitucional.buap.mx/server/api/core/bitstreams/a9c2fdb3-b018-45d6-a9f0-550c4e3a70cb/content

Huang, S., Wu, K., Chalamalasetti, S. R., El Hajj, I., Xu, C., Faraboschi, P., & Chen, D. (2021). A Python-based High-Level Programming Flow for CPU-FPGA Heterogeneous Systems : (Invited Paper). *Proceedings of PEHC 2021: Workshop on Programming Environments for Heterogeneous Computing, Held in Conjunction with SC 2021: The International Conference for High Performance Computing, Networking, Storage and Analysis*, 20–26. https://doi.org/10.1109/PEHC54839.2021.00008

Intel Corporation. (2020). *What Is Hyper-Threading?* https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html

Intel Corporation. (2021). *Intel DevCloud for oneAPI*. https://devcloud.intel.com/oneapi/get_started/

Intel Corporation. (2023, December 14). *Find the Shortest Path with a Floyd Warshall Algorithm SYCL* Implementation on GPU*. https://www.intel.com/content/www/us/en/developer/articles/technical/shortest-path-sycl-based-floyd-warshall-on-gpu.html

Lewis, N., & Hoffman, C. (2023, March 17). *CPU Basics: What Are Cores, Hyper-Threading, and Multiple CPUs?* https://www.howtogeek.com/194756/cpu-basics-multiple-cpus-cores-and-hyper-threading-explained/

Madiajagan, M., & Raj, S. S. (2019). Parallel Computing, Graphics Processing Unit (GPU) and New Hardware for Deep Learning in Computational Intelligence Research. *Deep Learning and Parallel Computing Environment for Bioengineering Systems*, 1–15. https://doi.org/10.1016/B978-0-12-816718-2.00008-7

Moreno, K. (2023, March 23). *Complejidad de un algoritmo (notación Big-O)*. Make It Real. https://guias.makeitreal.camp/docs/algoritmos/complejidad#complejidad-espacial

Naiouf, M., Chichizola, F., De Giusti, L., Montezanti, D., Rucci, E., & Frati, E. (2015). Arquitectura y Algoritmos Paralelos en HPC: Tendencias Actuales. *Instituto de Investigación En Informática LIDI (III-LIDI)*. http://sedici.unlp.edu.ar/handle/10915/46198

Naiouf, M., De Giusti, A. E., De Giusti, L. C., Chichizola, F., Sanz, V. M., Pousa, A., Rucci, E., Basgall, M. J., Sánchez, M., Costanzo, M., Gallo, S. L., Montes de Oca, E. S., Frati, F. E., & Gaudiani, A. A. (2021). Algoritmos paralelos y evaluación de rendimiento en plataformas de HPC. *XXIII Workshop de Investigadores En Ciencias de La Computación (WICC 2021, Chilecito, La Rioja)*, *August 2021*, 674–679. http://sedici.unlp.edu.ar/handle/10915/120328

Njoroge, H. (2022, October 31). *La ley de Amdahl*. LibreTexts. https://espanol.libretexts.org/Vocacional/Vocacional/Aplicaciones_inform%C3%A1ticas_y_tecnolog%C3%ADa_de_la_informaci%C3%B3n/Hardware_de_Tecnolog%C3%ADa_de_la_Informaci%C3%B3n/Arquitectura_Avanzada_de_Organizaci%C3%B3n_de_Computadoras_(Njoroge)/02%3A_Multiprocesamiento/2.01%3A_La_ley_de_Amdahl#title

Organización de las Naciones Unidas. (2018). *La Agenda 2030 y los Objetivos de Desarrollo Sostenible: una oportunidad para América Latina y el Caribe*. www.cepal.org/es/suscripciones

Padmanabhan, A. (2022, February 19). *Algorithmic Complexity*. DEVOPEDIA. https://devopedia.org/algorithmic-complexity

Ramírez Patiño, L. M., & Guerrero Hernández, L. E. (2017). *Arquitecturas Híbridas o Heterogéneas Paralelo entre NVIDIA CUDA y Parallel Studio XE para Intel® Xeon Phi™*. http://wiki.sc3.uis.edu.co/images/5/56/GR8.pdf

Reinders, J., Ashbaugh, B., Brodman, J., Kinsner, M., Pennycook, J., & Tian, X. (2021). Data Parallel C++. In *Data Parallel C++*. Apress. https://doi.org/10.1007/978-1-4842-5574-2

Requene, N. (2022). *IMPLEMENTACIÓN DE ALGORITMOS SOBRE ARQUITECTURA MULTINÚCLEO PARA OPTIMIZAR EL ALTO COSTE COMPUTACIONAL AL PROCESAR GRANDES VOLÚMENES DE DATOS* [UTN]. http://repositorio.utn.edu.ec/handle/123456789/13091

Rossainz López, M. (2020). *Programación Concurrente y Paralela*. https://www.cs.buap.mx/~rossainz/PCyP_Maestria/1_Apuntes/6_ProgParalela1.pdf

Soto, R. T. (2016). *Programación paralela sobre arquitecturas heterogéneas* [Universidad Nacional de Colombia]. https://repositorio.unal.edu.co/ handle/unal/57830

The multicore association, Levy, M., Stewart, D., & Domeika, M. (2013). *MULTICORE PROGRAMMING PRACTICES*. The Multicore Association. https://www.multicore-association.org/

Trimberger, S. M. S. (2018). Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology: This Paper Reflects on How Moore's Law Has Driven the Design of FPGAs Through Three Epochs: The Age of Invention, the Age of Expansion, and the Age of Accumulation. *IEEE Solid-State Circuits Magazine*, *10*(2), 16–29. https://doi.org/10.1109/MSSC.2018.2822862

Valdez, R., & Maldonado, Y. (2021). Computación heterogénea y FPGAs como aceleradores eficientes. *CICIC 2022 - Decima Segunda Conferencia Iberoamericana de Complejidad, Informatica y Cibernetica En El Contexto de the 13th International Multi-Conference on Complexity, Informatics, and Cybernetics, IMCIC 2022 - Memorias*, 25–29. https://doi.org/10.54808/CICIC2022.01.25

Weinbach, N. L. (2008). *Paradigmas de Programación en Paralelo: Paralelismo Explícito y Paralelismo Implícito* [Universidad Nacional del Sur]. http://repositoriodigital.uns.edu.ar/bitstream/123456789/2001/1/MgTesis%20Weinbach%20-%20Paradigmas%20de%20Programacion %20en%20Paralelo.pdf

ANEXOS

Anexo A: Guía Intel DevCloud

A continuación, se presenta una guía detallada acerca de la plataforma de Intel DevCloud, la cual se va a utilizar para desarrollar los algoritmos de programación heterogénea. Esta guía muestra paso a paso, como poder acceder a la herramienta de Intel oneAPI Base Toolkit, desde el primer paso que es crearnos una cuenta en Intel, hasta la conexión SSH para enlazar la terminal de nuestra computadora personal a los servidores de Intel.

Crear una cuenta en Intel DevCloud

Para la creación de una cuenta en Intel DevCloud se debe acceder a la plataforma oficial de Intel, se debe dar clic en el ícono de una persona que se encuentra en la parte superior de la página, como se muestra en la imagen.

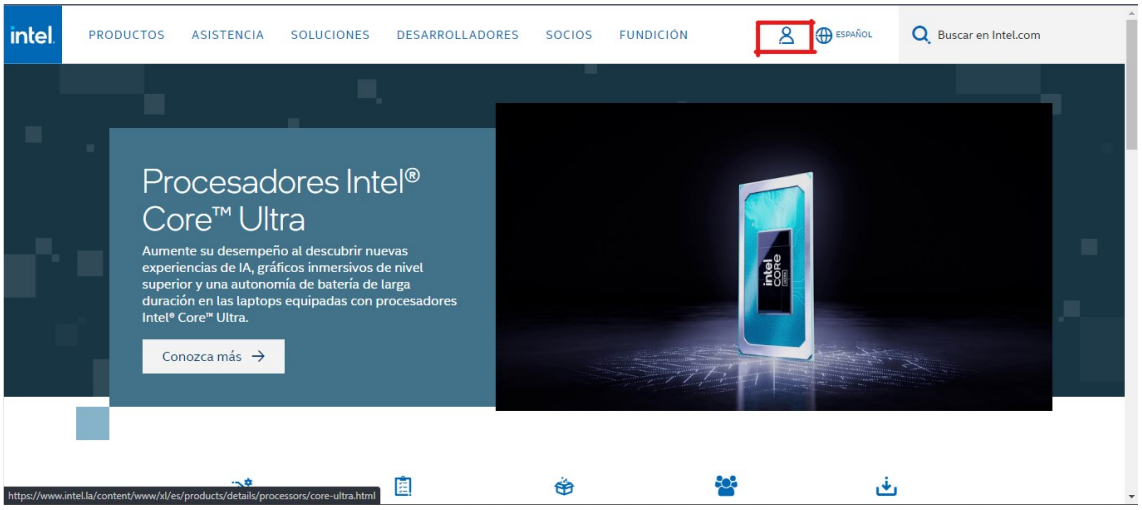


Figura 1: Página de inicio de Intel

Fuente: Elaboración propia

A continuación, nos llevará a la página de login, en la cual podemos ingresar a nuestra cuenta una vez hayamos creado con éxito nuestra cuenta de Intel, para ello debemos dar clic en el botón de Crear una cuenta.

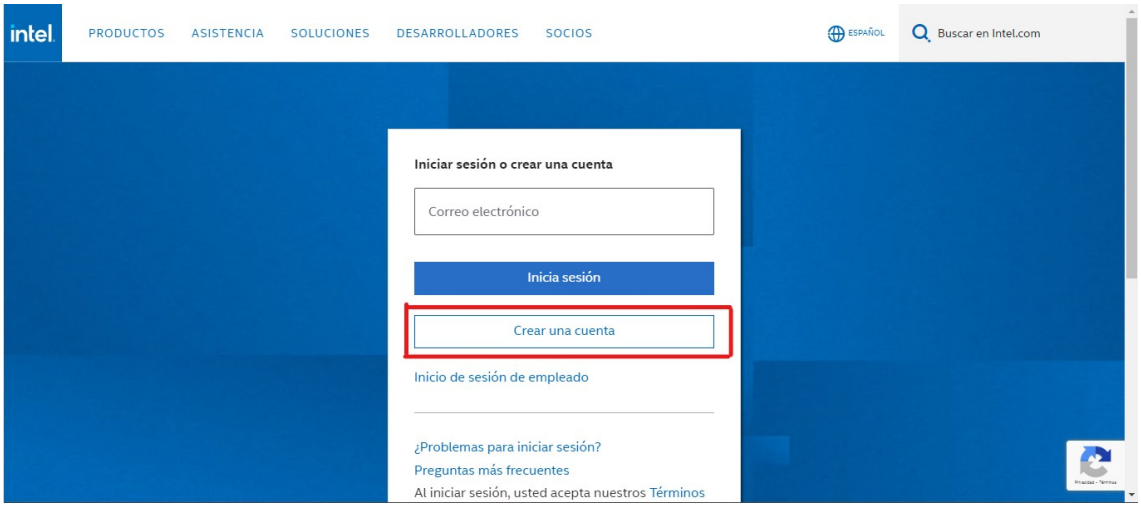


Figura 2: Página de Login de Intel

Fuente: Elaboración propia

En el siguiente apartado debemos llenar todos los campos con la información que nos solicita el formulario para la creación de una nueva cuenta. Una vez llenado todos los datos al botón de Verifique su correo electrónico.

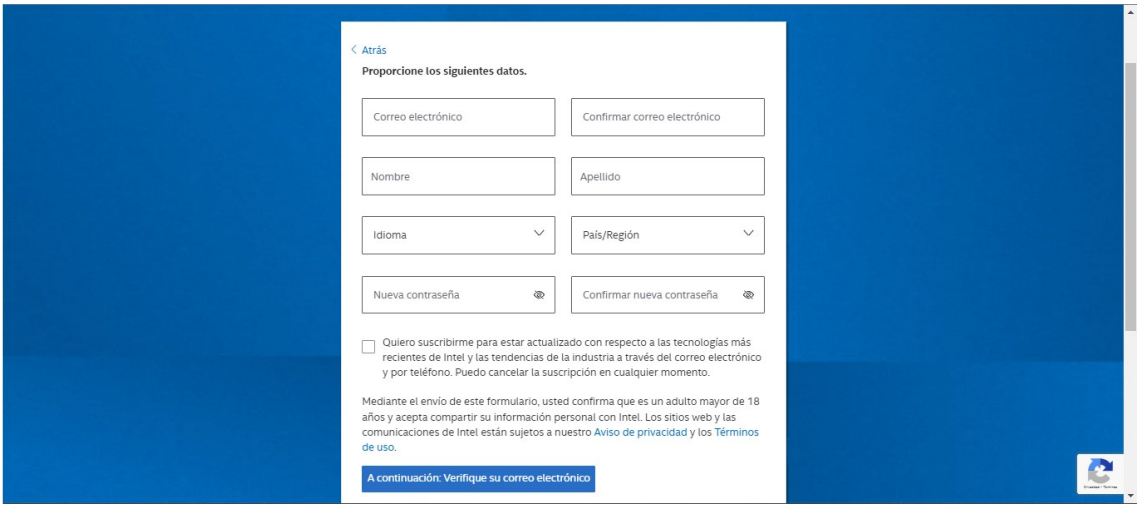


Figura 3: Formulario de registro de Intel

Fuente: Elaboración propia

Después nos enviará un código de verificación al correo que ingresamos en el formulario anterior, revisamos nuestro correo, en el caso de que no nos llegue el código verificación tenemos el botón de Enviar nuevo código. Si aun así no nos llega ningún código regresamos y verificamos que el correo que ingresamos sea el correcto. Una vez hayamos ingresado el código de verificación hacemos clic en el botón de Crear una cuenta.

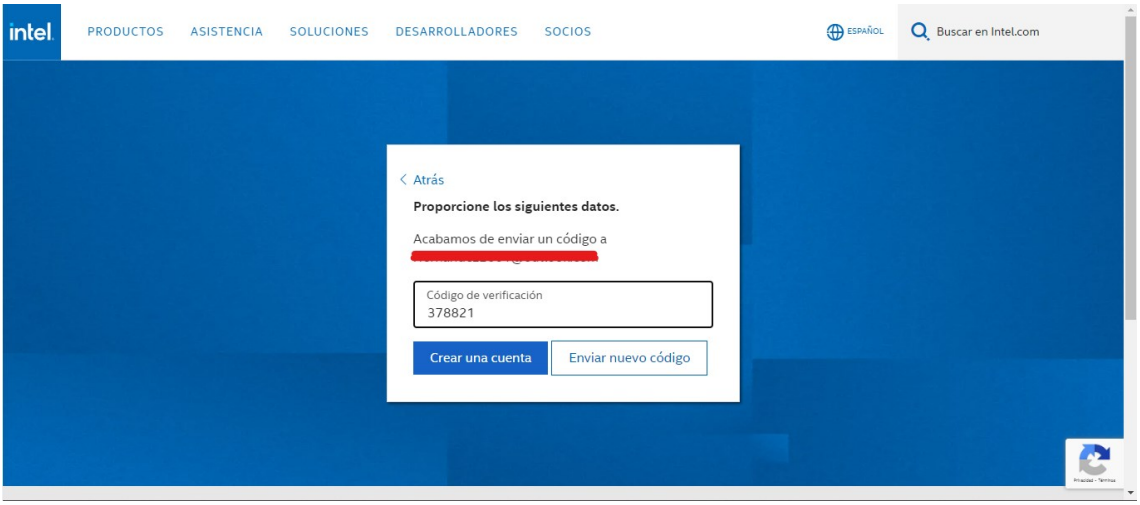


Figura 4: Ingresar código de verificación

Fuente: Elaboración propia

Al finalizar el último paso correctamente, su cuenta de Intel DevCloud ya está activa. Para comprobar que se ha iniciado sesión con su cuenta, se debe dar clic en el ícono de la persona que ahora tiene un visto dentro del ícono y se desplegará en Panel de su cuenta, además tendrá el botón de Desconectarse para cerrar sesión.

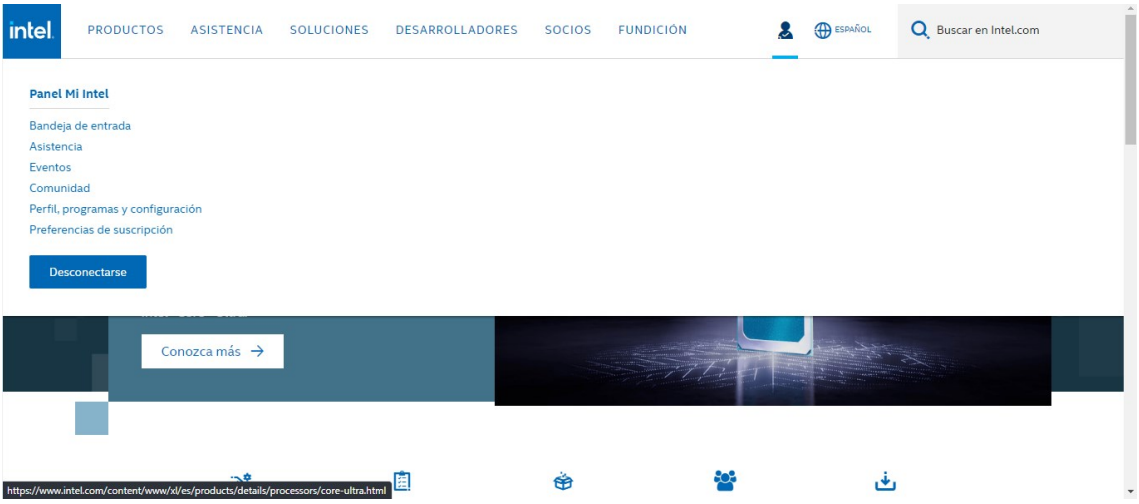


Figura 5: Panel Mi Intel

Fuente: Elaboración propia

Ingresar a oneAPI DevCloud

La plataforma de Intel tiene varias herramientas de desarrollo, una de ellas es oneAPI DevCloud la cual nos permite desarrollar código de programación heterogénea DPC++ y cuenta con una cantidad variada de hardware en la cual podemos ejecutar el código que hayamos desarrollado.

Para poder utilizar esta herramienta de desarrollo debemos ingresar al apartado de desarrolladores y después seleccionar Herramientas de desarrollo.

Figura 6: Intel: Herramientas de desarrollo.

Fuente: Elaboración propia

A continuación, en la sección de Get Software and Development Products debemos dar clic en Intel Developer Cloud.

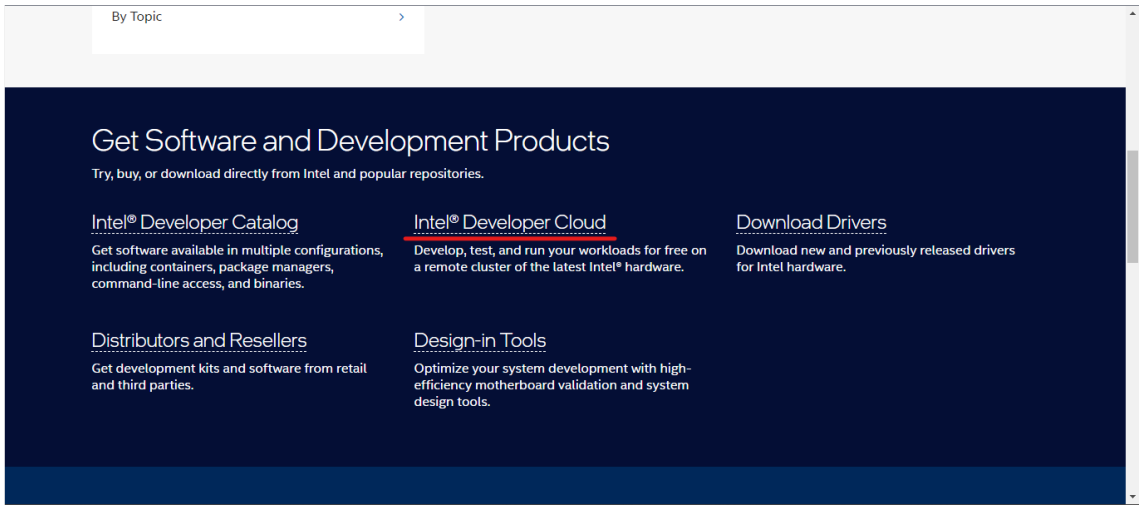


Figura 7: Get Software and Development Products (Intel Developer Cloud)

Fuente: Elaboración propia

Una vez ahí, nos mostrará tres opciones de herramientas que nos proporciona la plataforma de Intel, debemos escoger la tercera opción la cual es Build Multiarchitecture and FPGA Applications, podemos seleccionarla dando clic en el nombre o en el botón de Learn More.

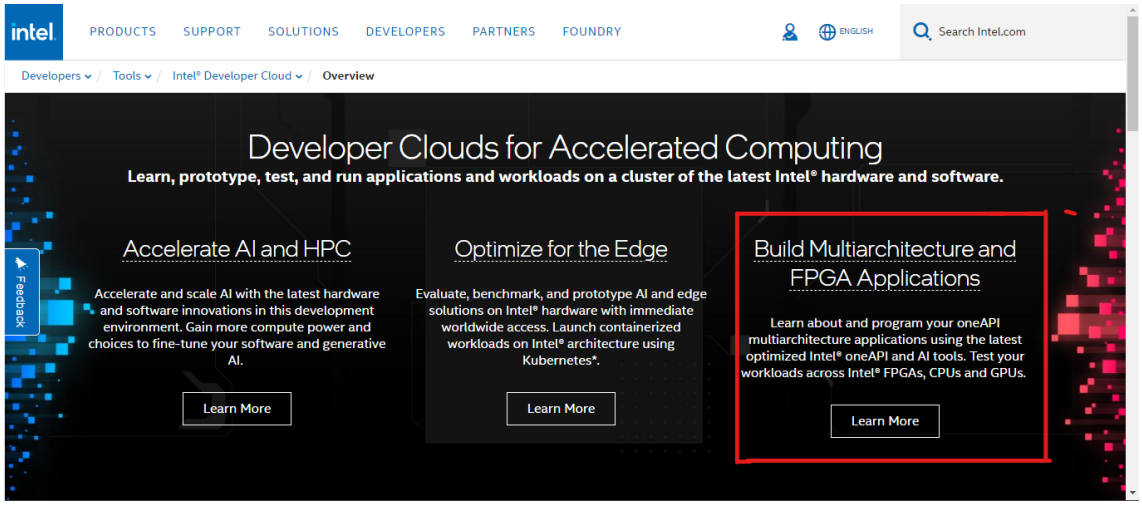


Figura 8: Intel: Build Multiarchitecture and FPGA Applications

Fuente: Elaboración propia

Si es la primera vez que ingresamos a este apartado, nos solicitará que realicemos la inscripción en Developer Cloud para OneAPI y poder acceder a la plataforma de desarrollo en la nube que nos proporciona Intel. Para ello debemos dar clic en el botón de Get Free Access.

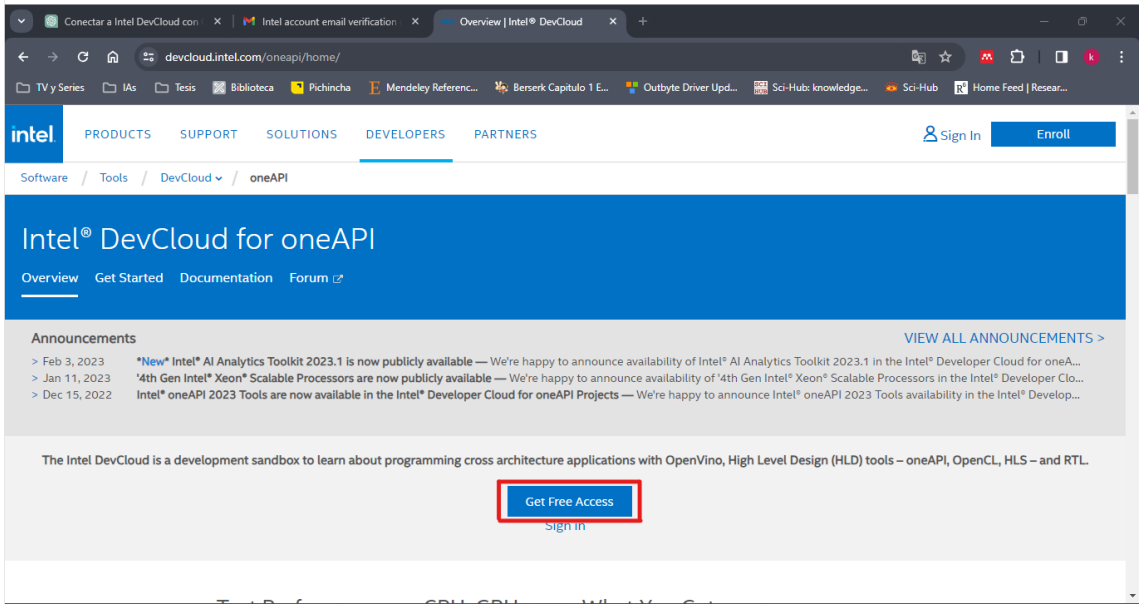


Figura 9: Ingresar por primera vez a Intel DevCloud for oneAPI

Fuente: Elaboración propia

A continuación, debemos ingresar la información que nos solicita para poder completar los pasos de la inscripción, es necesario aceptar los términos y condiciones. En la parte de Subscripciones de Comunicación no es necesario subscribirse a ninguna de las opciones.

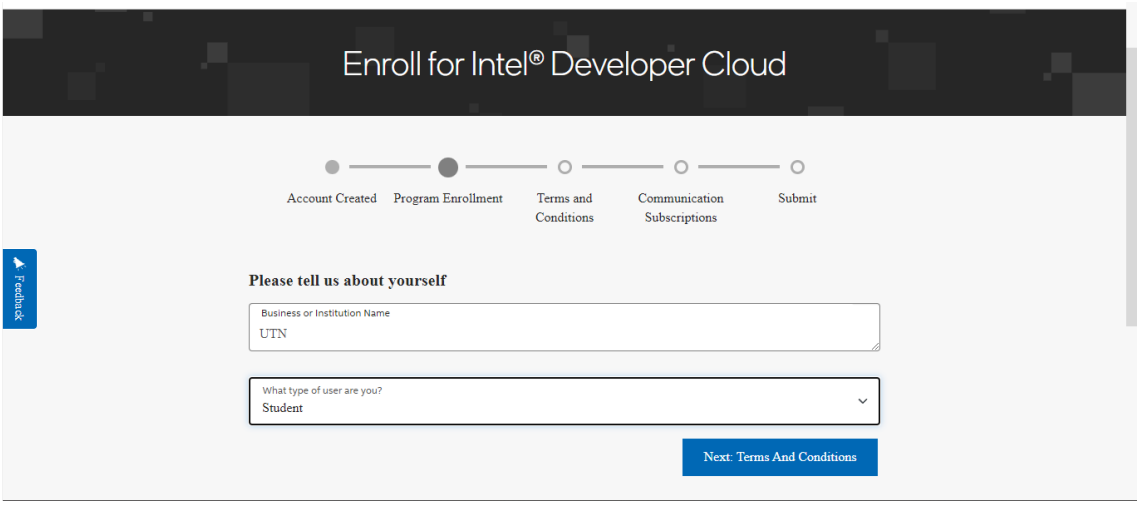


Figura 10: Inscripción a Intel Developer Cloud

Fuente: Elaboración propia

Una vez hayamos completado todos los pasos, nos mostrará la información de nuestra inscripción. Para finalizar con este proceso, debemos dar clic en el botón de Submit.

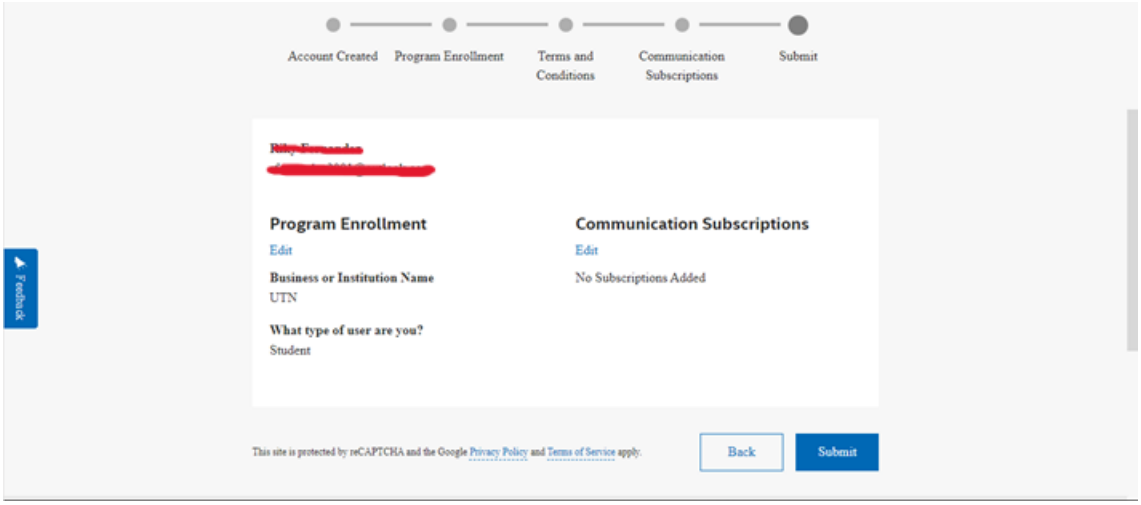


Figura 11: Datos de inscripción a Intel Developer Cloud

Fuente: Elaboración propia

Una vez ya hayas realizado todos los pasos te mostrará un mensaje que indica que ya estás inscrito en Intel Developer Cloud, para continuar debes hacer clic en el texto resaltado de azul que dice: Click here to complete provisioning your account.

Para el paso final de nuestra inscripción a Intel DevCloud, nos solicitará aceptar los términos y condiciones para utilizar todas las prestaciones que nos ofrece esta plataforma.

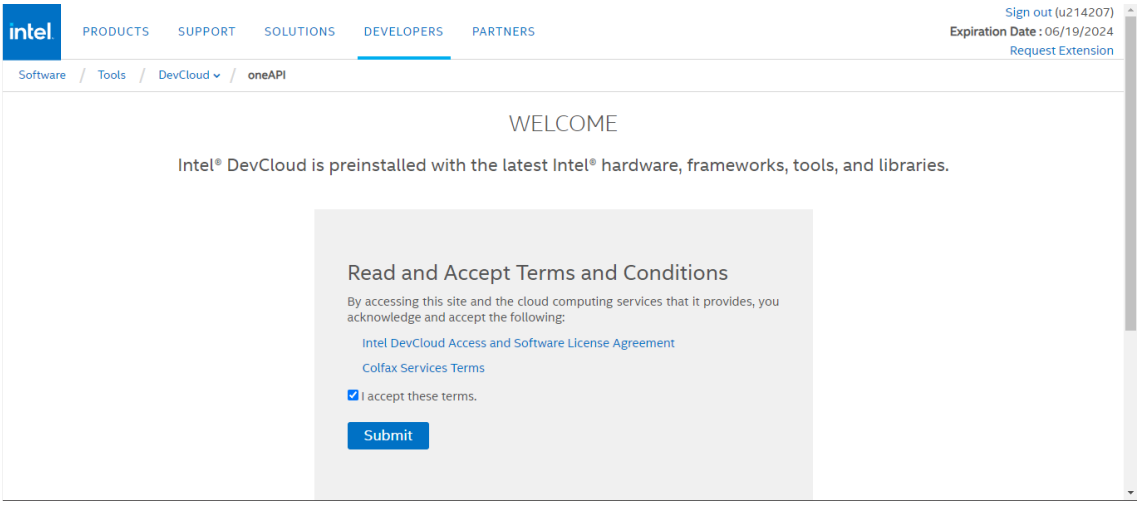


Figura 12: Términos y condiciones de Intel DevCloud

Fuente: Elaboración propia

Nota: Nuestra cuenta de Intel DevCloud expira en cuatro meses después del momento de nuestra inscripción. En el caso de necesitar más tiempo se puede solicitar una extensión para poder conservar los datos y no perder la cuenta.

Una vez que estemos inscritos en Intel DevCloud, podemos visualizar en la parte superior derecha el botón de cerrar sesión (Sign out), nuestro número de usuario, la fecha de expiración de nuestra cuenta y la opción de solicitar una extensión de nuestra cuenta (Request Extension).

En Overview tenemos una descripción general de lo que es Intel DevCloud for oneAPI, así como el hardware disponible, herramientas, Frameworks y librerías con las que cuenta esta plataforma. Además, existe la respectiva documentación de cómo utilizar todas las funcionalidades que nos ofrece Intel DevCloud para oneAPI. La plataforma también cuenta con un foro donde puedes postear preguntas para solventar dudas que tengas acerca de alguna herramienta o error que se te presente.

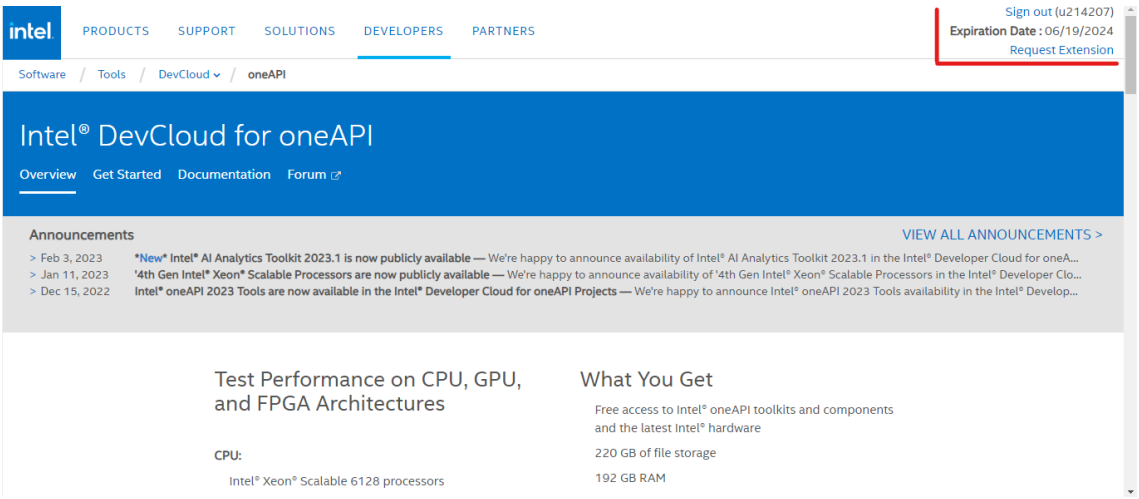


Figura 13: Descripción general de oneAPI

Fuente: Elaboración propia

Para poder utilizar oneAPI debemos ingresar a Get Started, una vez ahí tenemos varias herramientas (Toolkit), la opción que nos interesa es Intel oneAPI Base Toolkit. Es el software que nos permite desarrollar programación heterogénea mediante una conexión SSH desde nuestra computadora a los servidores de Intel.

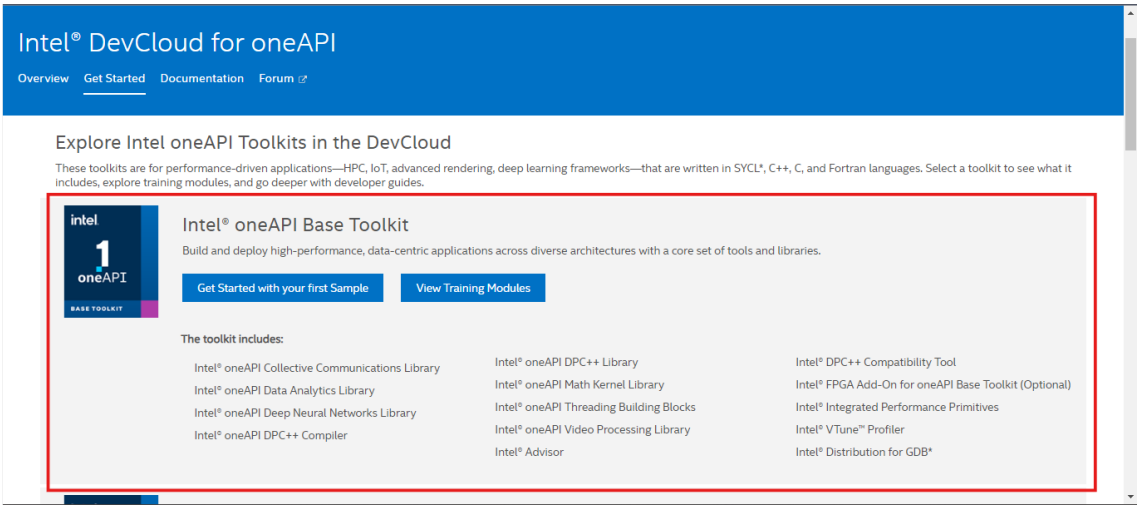


Figura 14: Intel oneAPI Base Toolkit

Fuente: Elaboración propia

Intel DevCloud también nos permite conectarnos mediante JupyterLab, que es la opción más rápida para empezar a utilizar Intel oneAPI Base Toolkit. Esta opción se localiza en Get Started, en la parte inferior de la página como se muestra en la siguiente figura, para ingresar debemos hacer clic en Launch JupyterLab* y se abrirá una nueva pestaña donde debemos iniciar sesión con el correo y contraseña que utilizamos para crear nuestra cuenta de Intel.

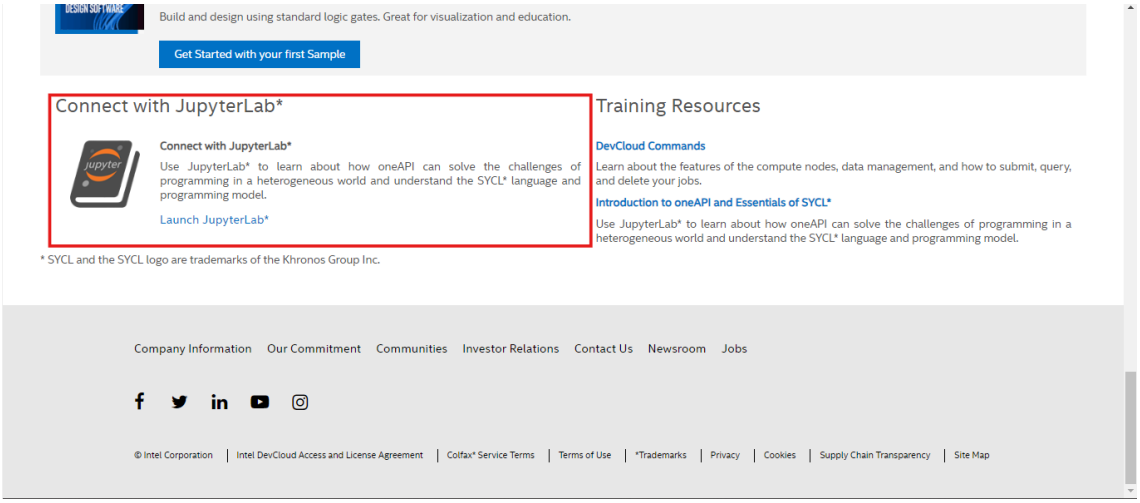


Figura 15: Conectarse a oneAPI con JupyterLab

Fuente: Elaboración propia

Una vez realizado todos los pasos anteriores correctamente, ya tenemos acceso a Intel DevCloud oneAPI. Al ingresar por primera vez a la plataforma es recomendable leer la guía de bienvenida para comprender como es el funcionamiento básico de la plataforma.

En la sección de la izquierda, se encuentran varios directorios que albergan ejemplos y tutoriales destinado a facilitar el aprendizaje de oneAPI. En particular, el directorio llamado **oneAPI_Essentials** se encuentran varios módulos que ofrecen información específica y detallada para el desarrollo de algoritmos heterogéneos utilizando lenguaje de programación DPC++.

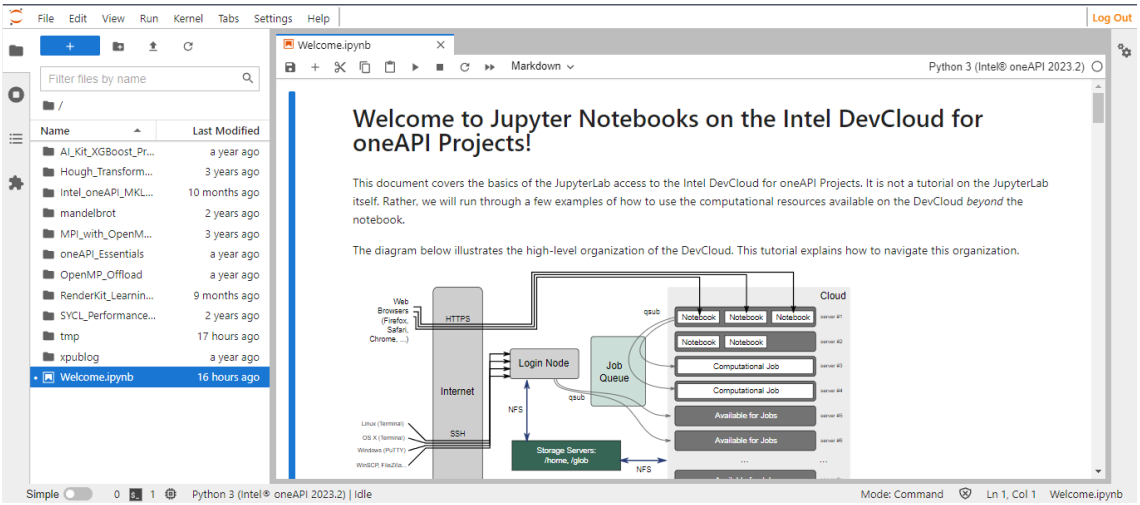


Figura 16: Intel oneAPI en JupyterLab

Fuente: Elaboración propia

Conectar una PC Windows a Intel DevCloud oneAPI por medio de SSH

La página oficial de Intel DevCloud cuenta con documentación y tutoriales para conectar DevCloud a diferentes sistemas operativos. Para acceder a dicho apartado debe ingresar a Get Started, en la sección de Intel oneAPI Base Toolkit dar clic en el botón de Get started with your first sample Figura: Intel oneAPI Base Toolkit, ahí se encuentran 3 opciones, debemos desplegar la primera opción que es Conectarse a DevCloud. Existen cuatro maneras de conectarse a DevCloud (Intel Corporation, 2021):

- OpenSSH en Windows (Recomendado).
- Cygwin en Windows (Obsoleto).
- Linux o macOS.
- Visual Studio Code.

A continuación, se detallarán los pasos que se siguieron para el desarrollo de este proyecto de tesis para conectar una terminal de Windows al Intel oneAPI Base Toolkit utilizando una conexión remota de SSH.

Para realizar la conexión de una PC con sistema operativo Windows es necesario la aplicación OpenSSH que habitualmente ya se encuentra instalada en nuestro sistema operativo por defecto (Intel Corporation, 2021). Si no es así, es necesario instalar el programa para continuar con los siguientes pasos de la guía.

Advertencia: Antes de avanzar, es importante que se comunique con el departamento de Tecnologías de la Información (TI) de su organización para obtener las configuraciones de PROXY_SERVER:PUERTO necesarias. La falta de esta información puede resultar en un fallo de la conexión remota a Intel DevCloud. Sin embargo, si se conecta directamente a Internet sin utilizar un proxy, puede ignorar esta advertencia (Intel Corporation, 2021).

Configuración de conexión SSH

Esta guía está basada en la guía oficial de Intel, se enfoca en resolver algunas dudas e inconvenientes que surgieron a lo largo del desarrollo de este proyecto de tesis.

1. Como primer paso, es necesario descargar la clave de acceso ssh de DevCloud de nuestro usuario en computadora. La clave de acceso se encuentra ingresando a Get started with your first sample como se muestra en la Figura: Intel oneAPI Base Toolkit, en la primera opción que es Connect to DevCloud, nos despliega las distintas formas de conectarnos a DevCloud. Debemos escoger cualquiera de las opciones exceptuando la opción de Visual Studio Code. Ahí en los primeros pasos

se encuentra el botón de SSH key for Linux/macOS/Cywin, al hacer clic en dicho botón se descargará automáticamente la clave.

2. Debemos abrir el directorio SSH de nuestra computadora la cual se encuentra en "C:\Users\user\.ssh" (Si no cuenta con la carpeta .ssh debe crearla manualmente). Una vez ahí, debemos pegar la clave que descargamos anteriormente.

3. Cree un archivo de texto llamado config sin la extensión .txt. Edite y pegue en el archivo el siguiente texto:

```
1 Host devcloud
2 User u******
3 IdentityFile ~/.ssh/devcloud-access-key-******.txt
4 ProxyCommand ssh -T -i ~/.ssh/devcloud-access-
5 key-******.txt guest@ssh.devcloud.intel.com
```

Nota: Debe reemplazar los asteriscos (*) con su número de usuario.

4. Si se encuentra en una red que enruta todo el tráfico a través de un servidor proxy, debe agregar las siguientes líneas al archivo config.

```
 1 Host devcloudx
 2 User u******
 3 Port 4022
 4 IdentityFile ~/.ssh/devcloud-access-key-******.txt
 5 ProxyCommand ssh -T devcloud-via-proxy
 6
 7 Host devcloud-via-proxy
 8 User guest
 9 Hostname ssh.devcloud.intel.com
10 IdentityFile ~/.ssh/devcloud-access-key-******.txt
11 LocalForward 4022 c009:22
12 ProxyCommand <path_to_ncat_exe> --proxy
13 PROXY_SERVER:PORT --proxy-type socks5 %h %p
```

Nota: PROXY_SERVER y PORT generalmente los proporciona su departamento de TI.

5. La combinación de OpenSSH y Windows Terminal ofrece una solución eficaz en el entorno de Windows. Con Windows Terminal, es posible establecer fácilmente una conexión a Intel DevCloud utilizando perfiles personalizados o de terceros, con tan solo un clic de un botón (Intel Corporation, 2021).

Para habilitar esta opción es necesario ingresar a la configuración de la Terminal como se muestra en la figura a continuación, una vez ahí debemos hacer clic en el botón de Abrir archivo JSON que se encuentra en la parte inferior izquierda del Terminal.

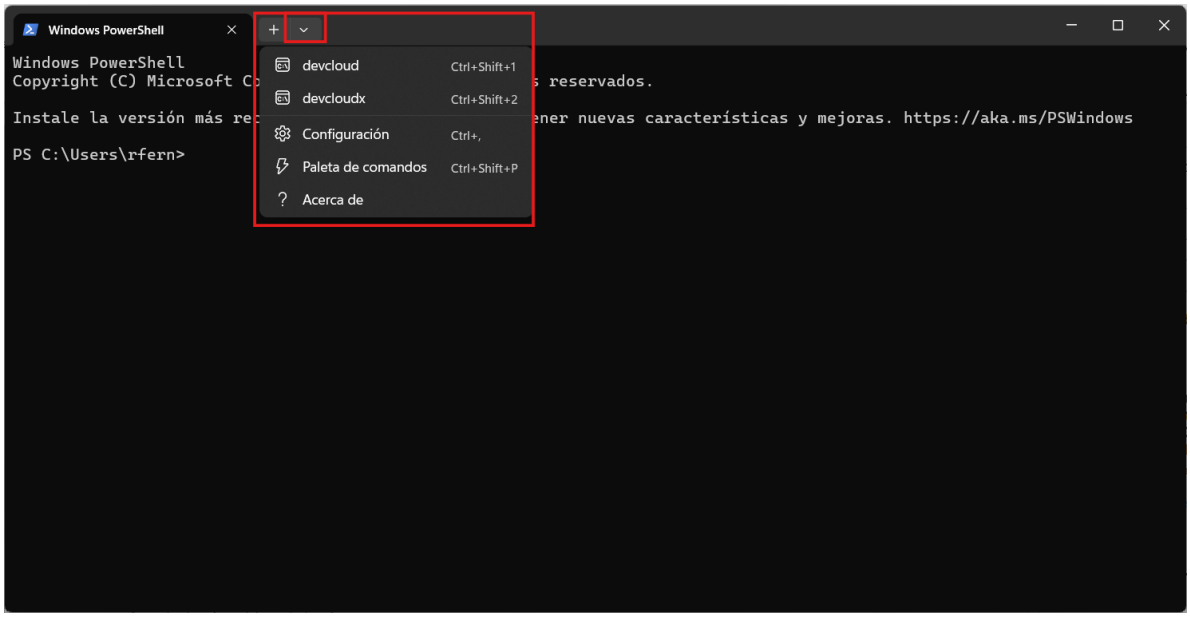


Figura 17: Terminal de Windows (Configuración)

Fuente: Elaboración propia

Se abrirá el archivo settings.json, podemos utilizar cualquier editor de texto para modificar el archivo. Debemos pegar los siguientes fragmentos JSON para crear los perfiles devcloud y devcloudx en nuestra terminal. Puede cambiar el guid a cualquier valor GUID (Intel Corporation, 2021).

Configuración de la conexión SSH directa a Intel DevCloud (devcloud):

```
{
    "commandline": "ssh devcloud",
    "guid": "{241213b4-e1a2-49ef-b8c0-22ae878f26b1}",
    "hidden": false,
    "name": "devcloud",
    "tabTitle": "DevCloud"
}
```

Configuración de la conexión SSH a Intel DevCloud a través de un servidor proxy (devcloudx).

```
{
    "commandline": "ssh devcloudx",
    "guid": "{14810d36-1f24-420d-9290-556321c0d7c6}",
    "hidden": false,
```

```
5       "name": "devcloudx",
6       "tabTitle": "DevCloud"
7   }
```

Figura 18: Archivo settings.json de la Terminal de Windows

Fuente: Elaboración propia

Para finalizar debemos guardar los cambios que realizamos. Una vez realzado los pasos anteriores, los perfiles estarán disponibles inmediatamente. Puede conectar con DevCloud seleccionando cualquiera de ellos o empleando los atajos de teclado ubicados a la derecha de cada elemento del menú. El perfil devcloudx solo se activará si su máquina local está conectada a Internet mediante un servidor proxy, mientras que el perfil devcloud establecerá la conexión cuando su máquina esté conectada directamente a Internet (Intel Corporation, 2021).

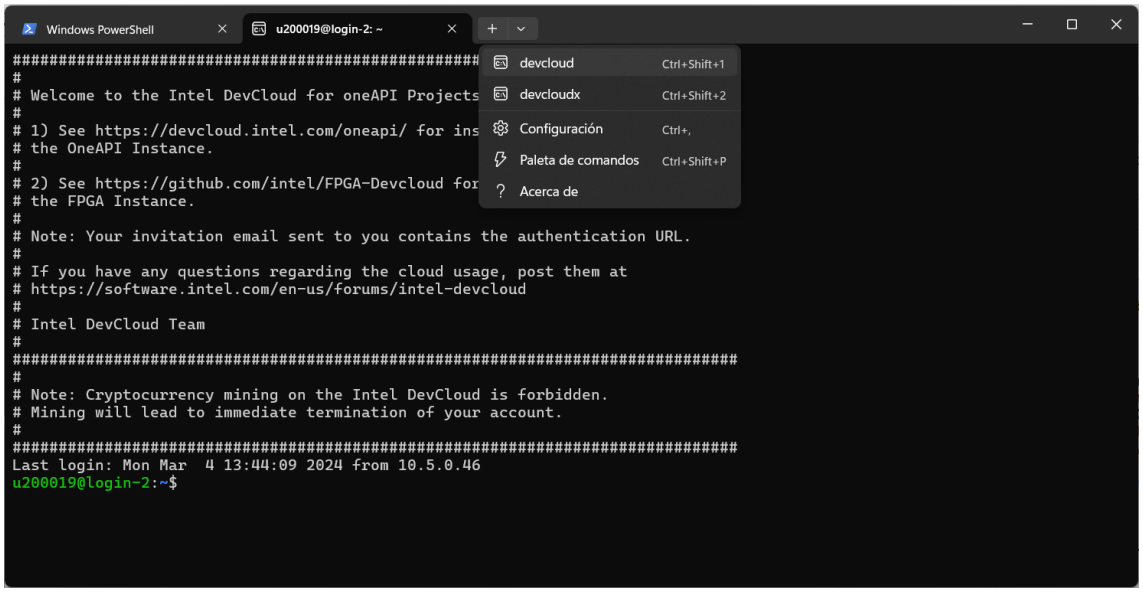


Figura 19: Conexión a Intel DevCloud oneAPI desde la Terminal

Fuente: Elaboración propia

Configurar conexión SSH de Visual Studio Code a Intel DevCloud oneAPI

Para realizar la conexión de Visual Studio Code a Intel DevCloud, es necesario completar la guía anterior de **Configuración de conexión SSH** para poder continuar con los siguientes pasos.

1. Es necesario realizar configuraciones adicionales para el acceso remoto utilizando Visual Studio Code. Para ello es necesario editar el archivo "%userprofile%\.ssh\config" con el siguiente texto:

```
# DevCloud VSCode config:
Host devcloud-vscode
UserKnownHostsFile /dev/null
StrictHostKeyChecking no
Hostname localhost
User u******
Port 5022
IdentityFile ~/.ssh/devcloud-access-key-******.txt

#SSH Tunnel config:
Host *.aidevcloud
User u******
IdentityFile ~/.ssh/devcloud-access-key-******.txt

# uncomment to enable the tunnel over the direct line
#ProxyCommand ssh -T devcloud nc %h %p

# uncomment to enable the tunnel over the proxy line
#ProxyCommand ssh -T devcloudx nc %h %p

LocalForward 5022 localhost:22
LocalForward 5901 localhost:590
```

Nota: Debe reemplazar los asteriscos (*) con su número de usuario.

2. Descargar e instalar Visual Studio Code en nuestra computadora.

3. Instalar la extensión Remote- SSH desde Visual Studio MarketPlace.

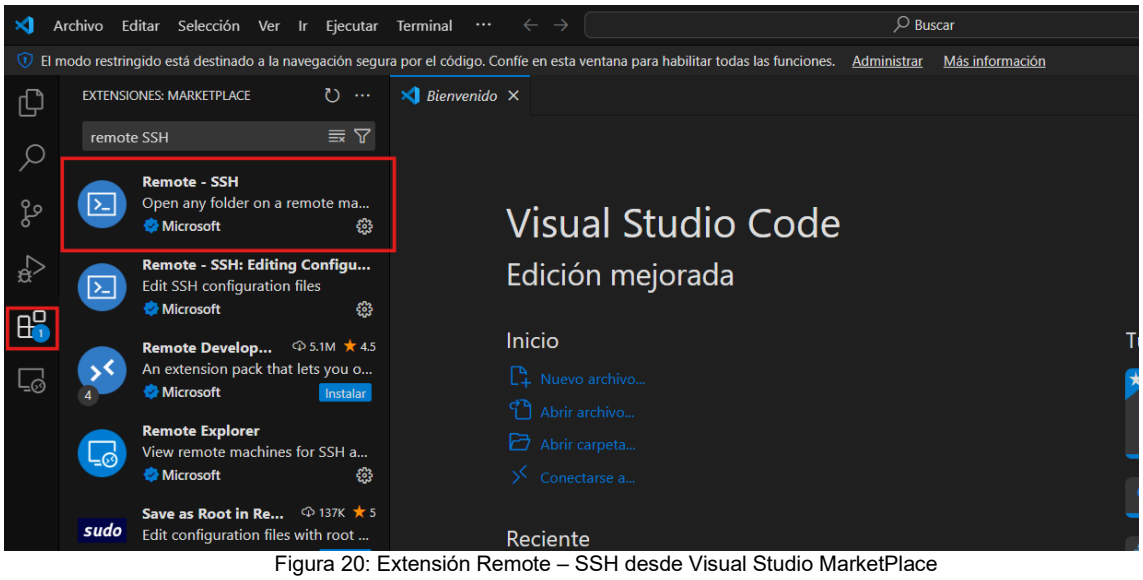


Figura 20: Extensión Remote – SSH desde Visual Studio MarketPlace

Fuente: Elaboración propia

Nota: Para el desarrollo de este proyecto se utilizó la versión 0.109.

4. Ingresar al Explorador remoto que se encuentra en la barra lateral izquierda. Ahí podemos observar las conexiones que configuramos en el archivo config en la guía anterior. La opción que nos interesa es devcloud-vscode, es la que nos permite conectar VS Code a Intel DevCloud oneAPI mediante túneles SSH.

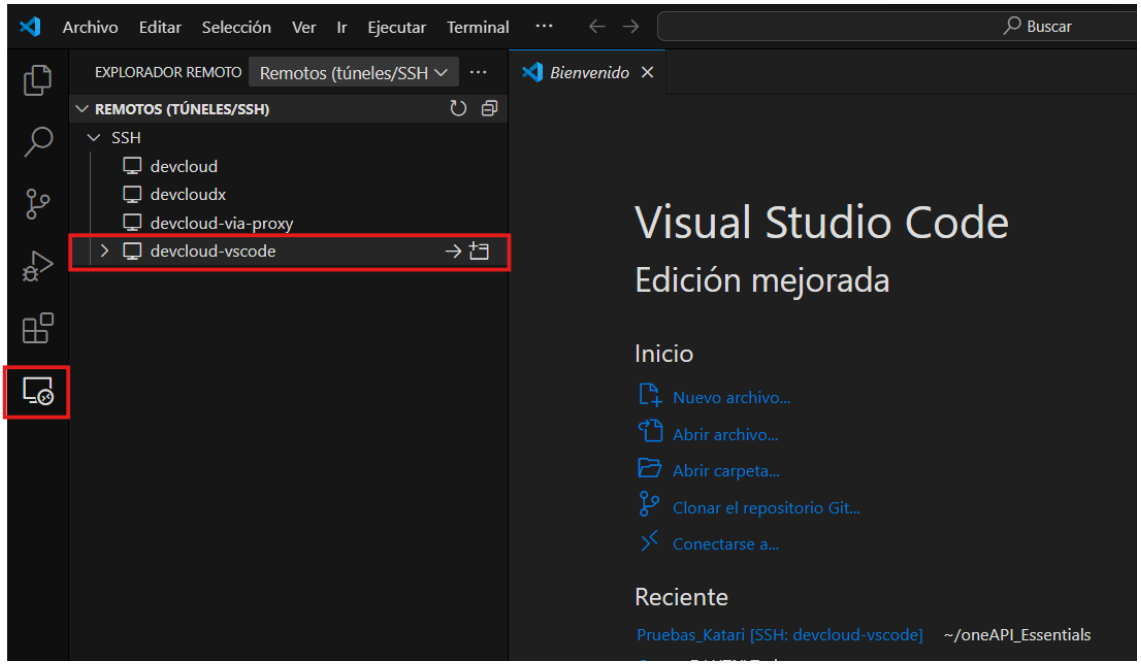


Figura 21: Conexión remota a DevCloud desde VS Code

Fuente: Elaboración propia

5. Configurar la conexión SSH para VS Code en un sistema operativo Windows.
   Ingresar a Archivo → Preferencias → Configuración
   En el buscador ingresar Remote.SSH. Ahí se debe configurar Config File y Path como se muestra en la siguiente figura.

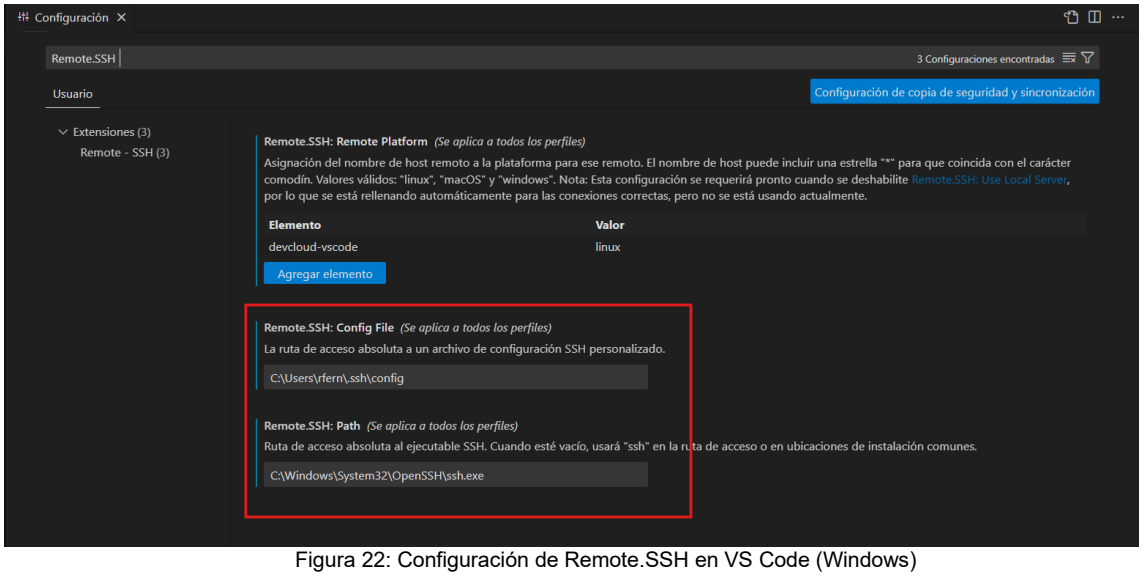


Figura 22: Configuración de Remote.SSH en VS Code (Windows)

Fuente: Elaboración propia

Conectarse a DevCloud desde Visual Studio Code

1. Conectarse a DevCloud desde la terminal de Windows utilizando los perfiles creados anteriormente “devcloud, devcloudx” dependiendo de su tipo conexión.

2. Solicitar un nodo de computación ingresando el siguiente comando:

```
u******@login-2:~$ qsub -I -l nodes=1:gpu:ppn=2 -d .
```

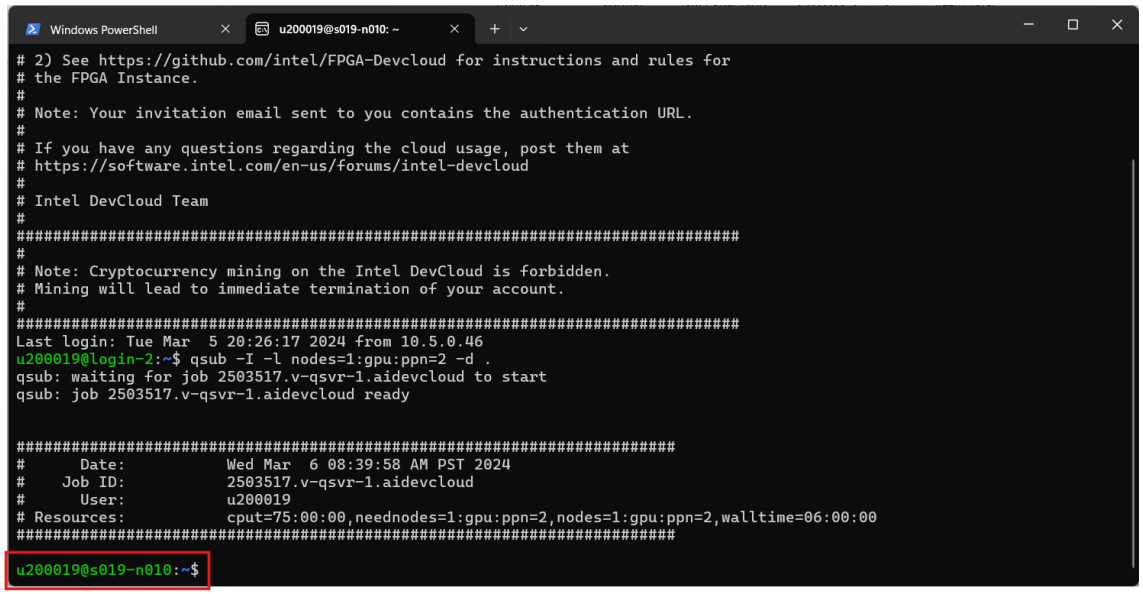


Figura 23: Conexión a un nodo de cómputo desde la Terminal

Fuente: Elaboración propia

Nota: Este comando de ejemplo solicita un nodo de cómputo que cuente con una GPU. Para conectarse a un nodo con otras especificaciones de hardware revise la documentación de Intel.

En la anterior figura podemos identificar que, al conectarnos a un nodo de cómputo, el prompt cambia, esta se compone de:

- Nombre de usuario: u200019
- Separador: “@”
- Nombre de host del nodo de cómputo: s019-n010.

3. Este paso solo lo realizamos una vez, dependiendo de nuestro tipo de conexión a la red.
   Es necesario configurar el archivo "C:\Users\user\.ssh\config". Asegúrese de que esté habilitado el ProxyCommand correcto para su conexión a Internet, es decir, un túnel a través de una conexión directa frente a un túnel a través de un proxy. En el siguiente ejemplo, ambas líneas de ProxyCommand están comentadas; descomente la que coincida con su caso.

```
1 ######################################################
2 # SSH Tunnel config
3 ######################################################
4 Host *.aidevcloud
5 User uXXXXX
6 IdentityFile ~/.ssh/devcloud-access-key-XXXXX.txt
7
8 # tunnel over the direct line
9 ProxyCommand ssh -T devcloud nc %h %p
10
11 # tunnel over the proxy line
12 #ProxyCommand ssh -T devcloudx nc %h %p
13
14 LocalForward 5022 localhost:22
15 LocalForward 5901 localhost:5901
16 ######################################################
```

4. Ejecutar el siguiente comando en una nueva terminal:

   ssh <compute-node-name>.aidevcloud

   Por ejemplo, si el nombre de host de su nodo de cómputo es s001-n059, ejecutaría el siguiente comando:

   ssh s001-n059.aidevcloud

5. Conecte VS Code a Intel DevCloud.
   Una vez realizado todos los pasos correctamente, solo nos queda conectar VS Code a Intel DevCloud utilizando la herramienta Remote SSH que instalamos previamente.

   Haga clic derecho en la línea devcloud-vscode, como se muestra a continuación:

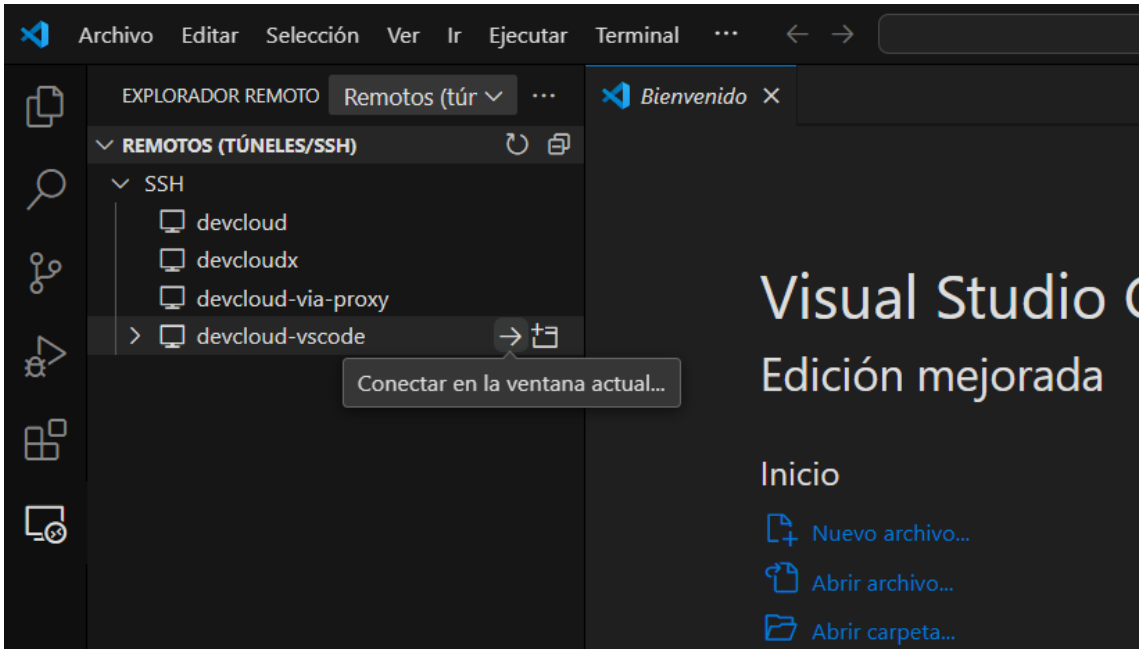


Figura 24: Conectar devcloud-vscode en la ventana actual

Fuente: Elaboración propia

Hacer clic en Conectar en ventana actual. VS Code establecerá una conexión con DevCloud que le permitirá buscar y abrir cualquier carpeta o archivo que tenga almacenado en su cuenta.

Ahora está conectado a Intel DevCloud. Puede verificar esto comprobando el nombre de configuración SSH, “SSH:devcloud-vscode”, en la esquina inferior izquierda.

Anexo B. Resultados en crudo de la comparativa de tiempos de procesamiento

En el siguiente enlace de GitHub se encuentran los tiempos de procesamiento obtenidos al ejecutar los algoritmos en las distintas arquitecturas. Además se encuentra adjunto los datasets utilizados para las pruebas de rendimiento del algoritmo de Matriz de Distancia Euclidiana.

Enlace: https://github.com/KatariTF/Tesis-programacion-heterogenea.git