Estudio de Viabilidad de SYCL como Modelo de Programación Unificado para Sistemas Heterogéneos Basados en GPUs en Bioinformática

Manuel Costanzo

Facultad de Informática
Universidad Nacional de La Plata

Director y Codirector (UNLP): Dr. Enzo Rucci y Dr. Marcelo Naiouf

Director (UCM): Dr. Carlos García Sanchez

Tesis presentada para obtener el grado de
Doctor en Ciencias Informáticas

Diciembre 2023

Resumen

Por un lado, la computación de alto rendimiento (HPC) requiere de modelos de programación que aprovechen el paralelismo masivo de arquitecturas heterogéneas modernas como las plataformas CPU-GPU. Sin embargo, modelos de programación como CUDA y OpenCL presentan limitaciones en portabilidad y productividad. SYCL surge como alternativa prometedora al unificar la programación en C++ y abstraer las particularidades del hardware. Por otro lado, la bioinformática y la biología computacional representan dos campos que han estado explotando las GPUs durante más de dos décadas, y muchas de sus implementaciones se basan en CUDA, lo que impone limitaciones significativas en cuanto a la portabilidad en una amplia gama de arquitecturas heterogéneas. Es por lo que esta tesis doctoral propone evaluar la viabilidad de SYCL como modelo de programación paralelo unificado, portable y eficiente para sistemas heterogéneos con GPUs, específicamente en bioinformática. Considerando que el alineamiento de secuencias biológicas representa una operación fundamental con amplias aplicaciones en diversas áreas de la biología y la medicina, se seleccionó la suite SW# como caso de estudio por su relevancia y por estar desarrollado en CUDA. Mediante la herramienta SYCLomatic se migró completamente el código de SW# de CUDA a SYCL. Este proceso involucró la ejecución de la herramienta, modificación del código generado, corrección de errores, verificación funcional, optimizaciones y estandarización SYCL. Posteriormente, se realizaron múltiples experimentos en un amplio conjunto de GPUs y CPUs de diferentes tipos y fabricantes para evaluar la portabilidad de rendimiento del código migrado en contextos individuales, tanto de GPU como de CPU, multi-GPU y CPU+GPU, en un entorno híbrido. Los resultados de esta tesis muestran, en primer lugar, que la herramienta SYCLomatic resulta efectiva y útil para la migración automática, aunque no pueda considerarse una solución final. En segundo lugar, las pruebas realizadas revelan que SYCL presenta una eficiencia comparable a la de CUDA en las GPUs NVIDIA y ofrece una amplia portabilidad funcional con una aceptable portabilidad de rendimiento a GPUs y CPUs de otros fabricantes. En conclusión, SYCL se posiciona como una alternativa viable como modelo unificado de programación heterogénea, portable y eficiente, estableciendo un precedente importante para futuros desarrollos en el área.

Índice general

Capítulo 1

Introducción

En primer lugar, se expone la motivación de esta tesis (Sección 1.1). Posteriormente, se detallan los objetivos y las metodologías a utilizar (Sección 1.2), así como los alcances y limitaciones (Sección 1.4), las contribuciones (Sección 1.5) y las publicaciones derivadas de esta investigación (Sección 1.6). Por último, se describe la organización del documento (Sección 1.7).

1.1. Motivación

La computación de alto rendimiento (HPC, por sus siglas en inglés) surge en las últimas décadas como una solución computacional esencial para abordar problemas que requieren alta demanda computacional como en la ciencia, la ingeniería y la industria [1]. Como resultado de los avances en arquitecturas de hardware paralelo y distribuido, así como en modelos de programación y algoritmos, el poder de cómputo de los sistemas HPC creció exponencialmente [2]. Por esta razón, la HPC actual se basa en sistemas que consisten en miles núcleos denominados *CUDA-cores* en las GPUs de NVIDIA que funcionan de manera coordinada, lo que representa un cambio de paradigma en comparación con la computación secuencial tradicional [3].

El mayor desafío en este nuevo paradigma es descubrir y aprovechar el enorme paralelismo brindado por estas modernas arquitecturas, para poder potenciar significativamente el funcionamiento y la eficiencia de las aplicaciones. Esto supone una reestructuración de los algoritmos y la creación de innovadoras estrategias de programación en paralelo, con la finalidad de distribuir de forma óptima el volumen de trabajo entre todos los recursos de procesamiento existentes [4].

Históricamente, el uso de la HPC se ha dado principalmente en el sector científico y computacional, donde la exigencia de un alto rendimiento proporciona razones para lidiar y superar su complejidad. No obstante, en los últimos años surgieron cambios debido a la aparición de arquitecturas *multicore* incluso en computadoras personales y *workstations*. Esta transición, acompañada de la creciente disponibilidad de aceleradores masivamente paralelos y especializados, como las Unidades de Procesamiento Gráficos (GPUs, por sus siglas en inglés), no sólo ofreció mayores capacidades computacionales, sino que también presentó ventajas en la eficiencia energética. La combinación de alto rendimiento con menor consumo energético favoreció tanto a la popularización de los aceleradores, como también a la expansión de la computación paralela hacia áreas como el aprendizaje automático, *Big Data*, computación en la nube, entre muchos otros [5].

Este avance significativo dio lugar a la aparición y aceptación de un modelo de computación heterogénea, en el que los sistemas integran procesadores *multicore* de uso genérico (CPU) con aceleradores especializados como GPUs, *Field Programmable Gate Arrays* (FPGAs), *Tensor Processing Units* (TPUs), entre otros [6]; todos con sus propias aptitudes y propiedades específicas. En particular, las GPUs se convirtieron en los aceleradores más popularmente adoptados en la actualidad, debido a su enorme capacidad para cómputo paralelo, sus prestaciones energéticas y su relación costo-beneficio [7]. Actualmente, los tres principales fabricantes son: NVIDIA, AMD e Intel, aunque la distribución del mercado no está equilibrada. En forma ilustrativa, para el segundo trimestre de 2023, Intel abarcaba el 3 % del mercado de GPUs discretas, mientras que AMD y NVIDIA el 10 % y 87 %, respectivamente. En cambio, al considerar las GPUs integradas, Intel cuenta con el 84 %, mientras que AMD con el 16 % [8].

Sin embargo, beneficiarse plenamente de este nuevo paradigma requiere más que solo avances arquitectónicos. Es esencial desarrollar software que pueda operar estas modernas y sofisticadas arquitecturas heterogéneas. En ese sentido, la comunidad en los últimos años ha fomentado la aparición de una gran variedad de modelos y lenguajes de programación diseñados específicamente para la computación heterogénea de alto rendimiento. Entre las opciones más destacadas se encuentran *Compute Unified Device Architecture* (CUDA) y *Open Computing Language* (OpenCL), que permiten aprovechar el paralelismo masivo en las GPUs, en el caso de CUDA, y en los aceleradores en general, en el caso de OpenCL. También existen extensiones de lenguajes tradicionales como *Open Multi-Processing* (OpenMP) para CPUs y GPUs y *Open Accelerators* (OpenACC) que brindan compatibilidad con múltiples GPUs. Además, el desarrollo de *frameworks* como *Kokkos* [1] y *RAJA* [2] surgen como respuesta a la necesidad de abstracciones de programación más flexibles y portables, permitiendo así una mayor eficiencia y escalabilidad en diversas plataformas de hardware.

Por su parte, CUDA [9] representa un modelo de programación y un entorno operativo creado por NVIDIA para resolver problemas de propósito general en sus GPUs. Introducido en 2007, CUDA extendió los lenguajes C/C++ con componentes para manifestar paralelismo a gran escala, posibilitando la creación de aplicaciones altamente eficientes para GPUs de NVIDIA. Con el tiempo, CUDA se estableció como el lenguaje predominante para la programación de GPUs y es frecuentemente empleado en HPC [10]. De hecho, en el 2019, encontramos más de 600 aplicaciones HPC en diversos dominios empleando CUDA, cifra que fue aumentando hasta la fecha [3]. Sin embargo, dado que es un modelo propietario, sólo es compatible con GPUs de NVIDIA, lo cual limita significativamente a la portabilidad del código para otras arquitecturas e incluso para GPUs de otros fabricantes.

Por otro lado, OpenCL [11] es un estándar para la programación en paralelo en plataformas heterogéneas, mantenido por el grupo Khronos. Ofrece un modelo de programación basado en C99 que facilita la expresión de paralelismo, posibilitando la ejecución de un mismo código fuente en CPUs, GPUs, FPGAs y otros aceleradores. Aún cuando OpenCL se puede adaptar a otras arquitecturas a nivel del código fuente, alcanzar una eficiencia óptima de manera portátil es muy complicado, ya que se necesitan optimizaciones específicas por cada plataforma. Adicionalmente, no está tan maduro ni se utiliza tanto como CUDA, resultando su programación más verbosa [12].

Alternativamente, OpenMP [13] y OpenACC [14, 15] representan dos enfoques distintos para abordar la programación paralela y heterogénea. Inicialmente, OpenMP se enfocaba en la paralelización en CPUs, pero ha evolucionado para soportar GPUs y otros aceleradores.

[1] `https://github.com/kokkos/kokkos`
[2] `https://github.com/LLNL/RAJA`
[3] `https://developer.nvidia.com/blog/nvidia-announces-cuda-x-hpc/`

Sin embargo, a pesar de su simplicidad, la adaptabilidad en plataformas heterogéneas es aún un trabajo en progreso, no logrando ofrecer la misma flexibilidad y control que se consigue con OpenCL o CUDA. Por su parte, OpenACC es similar en naturaleza a OpenMP, pero está específicamente diseñado para la computación heterogénea. Lamentablemente, su adopción ha sido limitada debido a que usualmente se requiere de ajustes específicos y su soporte por parte de los proveedores de herramientas no es universal.

En el contexto de estas herramientas y lenguajes, surgen Kokkos y RAJA como *frameworks* para la programación heterogénea. Kokkos es un modelo de programación en C++ diseñado para escribir aplicaciones portables para diversas arquitecturas, incluyendo CPUs, GPUs y otros aceleradores. Proporciona abstracciones que permiten un desarrollo eficiente y adaptativo a las características específicas de cada hardware. Por otro lado, RAJA, también basado en C++, se enfoca en la abstracción de bucles para lograr un rendimiento óptimo en plataformas heterogéneas.

A pesar de los innegables progresos que estos modelos de programación heterogénea han aportado, la realidad marca que la mayoría todavía tienen notables limitaciones que obstaculizan seriamente su adopción generalizada. Entre estas restricciones se incluyen su limitada portabilidad, ya que a menudo están reducidos a fabricantes o arquitecturas específicos (como CUDA), la complejidad de su curva de aprendizaje y la modesta productividad que proporcionan a los programadores (como OpenCL) [16].

Debido a estas restricciones pendientes, la comunidad HPC empezó en los últimos años a indagar en la aparición y aceptación de nuevos modelos y soluciones de programación paralela que sean al mismo tiempo más sencillos, portables y eficientes. El objetivo es popularizar el poder de la computación heterogénea, facilitando su uso tanto en aplicaciones científicas convencionales, como también en la creciente necesidad de aplicaciones de cálculo intensivo. Por ello, el concepto de portabilidad del rendimiento se convierte en esencial, el cual contempla dos aspectos claves: (1) hacer posible la ejecución de una única aplicación en diferentes arquitecturas de hardware y (2) alcanzar un nivel de rendimiento esperado en estas plataformas.

En este escenario, SYCL [17] surge recientemente como una opción especialmente prometedora para convertirse en un estándar unificado para la programación heterogénea paralela [4]. SYCL es un modelo de programación de alto nivel basado en C++ estándar, que da la posibilidad a los programadores de crear código portable, que pueda ejecutarse de manera eficaz en arquitecturas heterogéneas que combinan CPUs con diversos tipos de aceleradores. Esto se logra a través de la separación de las particularidades de cada arquitectura hardware mediante la aplicación extensiva de *templates* y abstracciones en C++ moderno. Por otro lado, SYCL está diseñado para ofrecer un alto rendimiento general, permitiendo a los programadores, si es necesario, explotar características particulares de cada hardware e integrar código OpenCL o nativo optimizado a bajo nivel cuando sea requerido, aprovechando la compatibilidad de ambos modelos [18].

Hoy en día, existen varias implementaciones de SYCL disponibles, como ComputeCpp [19] de Codeplay, oneAPI de Intel [20], triSYCL [21] liderado por Xilinx y AdaptiveCpp [22] (anteriormente llamada hipSYCL), coordinado por la Universidad de Heidelberg. Entre todos estos, oneAPI se destaca como la suite de desarrollo más avanzada, gracias a su naturaleza de proyecto abierto y multi-industria, diseñado para ofrecer un modelo de programación de alto rendimiento. Principalmente, oneAPI incluye el lenguaje *Data Parallel C++* (DPC++), que soporta los estándares SYCL y OpenCL sin necesidad de extensiones adicionales, y promueve la interoperabilidad con librerías optimizadas como oneCCL, oneDAL, oneDNN, oneMKL, oneTBB y oneVPL, satisfaciendo las necesidades de

[4] `https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf`

diversos dominios de aplicaciones paralelas [23].

Debido al gran número de aplicaciones HPC que se desarrollaron en lenguajes y modelos más antiguos como CUDA [10] y OpenCL, se precisa el desarrollo de técnicas efectivas para la migración de estos códigos tradicionales a SYCL. Por un lado, SYCL ofrece una interoperabilidad nativa con OpenCL [5], lo que facilita la integración de código existente de este modelo dentro de aplicaciones SYCL. De esta manera, no sólo se aprovechan las ventajas de portabilidad y productividad que ofrece SYCL, sino que también se respeta el código ya establecido en OpenCL, ofreciendo una transición más suave. Por otro lado, herramientas de portabilidad como `SYCLomatic` [24] provistas por oneAPI, están empezando a jugar un papel central en este proceso, al simplificar la automatización de la conversión de código de CUDA a SYCL. Aunque en general se necesita intervención manual adicional para finalizar la migración o para incrementar el rendimiento, estas herramientas suelen reducir bastante la carga de trabajo que se requiere.

Entre las áreas que potencialmente podrían aprovechar los beneficios de SYCL se encuentra la Bioinformática, ya que ha estado explotando el uso de aceleradores como las GPUs por más de dos décadas [7]. La bioinformática es un campo que surge del avance de tecnologías de información y de la evolución de técnicas cada vez más veloces para la secuenciación del ácido desoxirribonucleico (ADN). Está estrechamente relacionada con disciplinas como biología, bioquímica, informática, matemáticas, física y estadística y su finalidad es buscar y descubrir información relevante oculta en grandes volúmenes de datos, para obtener una mejor comprensión de la biología de los organismos [25].

Son múltiples los desafíos biológicos que requieren de un alto procesamiento computacional y es por lo que existen numerosas aplicaciones de GPU para enfrentarlos, como el alineamiento de secuencias [26], la dinámica molecular [27], el acoplamiento molecular [28], la predicción y búsqueda de estructuras moleculares [29], el análisis filogenético [30], entre otros ejemplos. En particular, el alineamiento de secuencias biológicas (ASB) desempeña un papel fundamental en la investigación biológica, ya que se utiliza en la mayoría de sus disciplinas. El objetivo del alineamiento es comparar dos o más secuencias biológicas para identificar las regiones que comparten una historia evolutiva común. Por esta razón, existen varias aplicaciones destacadas como CUDAlign [31, 32, 33, 34, 35], SW# [36], SW#db [37], CUDASW [38, 39, 40, 41], ADEPT [42], entre otras. Sin embargo, debido a que en general se han desarrollado sobre CUDA, la portabilidad a otras arquitecturas se ve gravemente limitada, lo que restringe explotar las ventajas que pueden presentar los diferentes dispositivos actuales o emergentes. Por lo tanto, una transición eficiente de aplicaciones bioinformáticas hacia entornos de programación como SYCL representaría un avance significativo, simplificando el desarrollo de nuevas aplicaciones y prolongando la relevancia y el alcance de las ya establecidas.

Conforme a este contexto y motivación, el objetivo principal de esta tesis doctoral consiste en analizar minuciosamente la factibilidad de emplear SYCL como un modelo unificado de programación paralela, que sea portable y al mismo tiempo eficiente, para el desarrollo y ejecución de aplicaciones que requieran alta demanda computacional en sistemas heterogéneos basados en GPUs, específicamente en el campo de la bioinformática.

[5] `https://www.intel.com/content/www/us/en/developer/articles/technical/sycl-interoperability-study-opencl-kernel-in-dpc.html`

1.2. Objetivos

El objetivo general de esta tesis consiste en evaluar la viabilidad de SYCL como un modelo de programación heterogénea unificado, portable y eficiente para el diseño y desarrollo de aplicaciones con alta demanda computacional en sistemas heterogéneos basados en GPUs, específicamente en el ámbito de la bioinformática.

Los objetivos específicos son:

- Investigar y comparar críticamente los modelos de programación y métricas de rendimiento presentes en el contexto de la computación heterogénea, específicamente en aplicaciones bioinformáticas, con el propósito de establecer una base conceptual para esta investigación.
- Diseñar y desarrollar software que aproveche las capacidades de SYCL para sistemas heterogéneos basados en GPUs en el contexto de la bioinformática, considerando especialmente la migración de aplicaciones implementadas en CUDA.
- Medir y comparar las prestaciones del software desarrollado en distintos sistemas heterogéneos basados en GPUs, considerando portabilidad, rendimiento y productividad como parámetros de interés.

1.3. Metodología

Esta investigación doctoral se plantea como un estudio proyectivo que pretende contribuir con la evaluación de SYCL como una alternativa para sistemas heterogéneos basados en GPUs, en el contexto de la bioinformática, más precisamente en el ASB. Para poder cumplir con los diferentes objetivos se realizaron las siguientes actividades:

- Se llevó a cabo un relevamiento de la bibliografía existente en la temática a partir de publicaciones científicas en bases de datos especializadas, libros y literatura gris.
- Se seleccionó el ASB como caso de estudio debido a que es una operación fundamental en bioinformática con amplias aplicaciones en diversas áreas de la biología. El ASB consiste en comparar dos o más secuencias biológicas, permitiendo identificar regiones conservadas, revelar relaciones evolutivas y funcionales, así como predecir la función de secuencias desconocidas. Además, es esencial para estudiar la evolución de especies, identificar genes homólogos y reconstruir la historia evolutiva. En el ámbito médico, el alineamiento de secuencias facilita el diseño de fármacos y la investigación de enfermedades. En resumen, el alineamiento de secuencias es una herramienta esencial en bioinformática que facilita el análisis y comprensión de las secuencias biológicas.
- Se llevó a cabo un estudio de diversas implementaciones de ASB. Se priorizó una implementación representativa de las necesidades y desafíos actuales en el campo de la bioinformática, como es el ASB, de manera eficiente. Además, era fundamental que esta implementación estuviera originalmente desarrollada en CUDA, con el fin de facilitar un estudio detallado sobre la migración y adaptación a arquitecturas heterogéneas y modernas. La implementación seleccionada se utilizó como caso de estudio en el desarrollo de la investigación.
- Se instalaron las herramientas necesarias para desarrollar y ejecutar aplicaciones en SYCL, incluyendo compiladores, librerías y herramientas de perfilado y depuración, en varios entornos de HPC.

- Se diseñó un flujo de trabajo y un conjunto de buenas prácticas para migrar código SYCL, que incluyó el uso de herramientas de automatización, técnicas de perfilado y optimizaciones manuales.
- Se migró una aplicación bioinformática completamente, siguiendo el flujo de trabajo y las buenas prácticas diseñadas en el paso anterior.
- Se midió el rendimiento de la versión migrada en SYCL, comparándola con la versión original en CUDA. Se evaluó la portabilidad de rendimiento de la versión SYCL en una amplia variedad de plataformas HPC con diferentes tipos de procesadores CPU/GPU de diversos fabricantes, como también en diferentes implementaciones SYCL.
- Se compararon los resultados obtenidos con los de otros estudios de contexto similar. Se analizaron tanto los resultados de la migración como el rendimiento y la portabilidad de la implementación seleccionada.
- Se realizó un análisis integral de los resultados obtenidos, evaluando la eficacia del código SYCL en términos de rendimiento, portabilidad y productividad del desarrollo.
- Finalmente, se difundió el código fuente de la versión SYCL migrada y optimizada a través de repositorios públicos, lo que promueve su reutilización por parte de la comunidad bioinformática y HPC en general.

1.4. Alcances y limitaciones

En el contexto de esta tesis, el enfoque se centró en la viabilidad y eficacia de SYCL como modelo de programación para sistemas heterogéneos basados en GPU, especialmente en aplicaciones bioinformáticas. La elección de SYCL responde a su promesa de unificación y portabilidad en distintas arquitecturas de hardware. Esto incluye la adaptación y evaluación de aplicaciones existentes, especialmente en el campo del alineamiento de secuencias, un área crítica en bioinformática, que fue seleccionada por su relevancia y desafío computacional, reconociendo que existen otras aplicaciones en bioinformática que podrían comportarse diferente bajo el modelo de SYCL.

Respecto a la exploración de sistemas heterogéneos, se optó por una diversidad de GPUs, abarcando distintas configuraciones y especificaciones, que tiene como fundamento la prevalencia y el impacto significativo de las GPUs en el rendimiento del HPC, como se detalla en la Sección 2.1.2. Sin embargo, aunque en esta investigación se utilizó un conjunto amplio y diverso de GPUs y CPUs de distintos fabricantes, es importante reconocer que el estudio no es exhaustivo en términos de cobertura de absolutamente todas las posibles configuraciones de GPU en el mercado, lo cual representa una limitación en términos de generalización de los resultados.

Adicionalmente, en esta investigación el enfoque principal se centró en la migración de código heredado de CUDA a SYCL, explorando así la adaptabilidad y funcionalidad de este último en el contexto de la computación heterogénea. Al priorizar este enfoque, se descartó la implementación de códigos nuevos desde cero en SYCL, como también la optimización del código migrado. Esta delimitación del alcance, aunque estratégica para evaluar la compatibilidad y la portabilidad entre CUDA y SYCL, limita la comprensión de las potenciales optimizaciones que SYCL ofrece.

En este estudio, se ha optado por centrarse en SYCL y excluir otros *frameworks* y lenguajes de programación en el campo de la computación heterogénea, como OpenCL, Kokkos

o Raja, así como lenguajes específicos para otras arquitecturas de hardware. Esta elección se debe tanto a una decisión deliberada como a consideraciones de alcance y delimitación del estudio. Aunque el rendimiento y la portabilidad han sido aspectos fundamentales en nuestra evaluación, reconocemos que la inclusión de estos *frameworks* y la consideración de otros aspectos de prestaciones (como la eficiencia energética) habrían enriquecido la comprensión general de la portabilidad de rendimiento en el ámbito de HPC. Sin embargo, estas extensiones se encuentran fuera de los límites establecidos para esta investigación, lo que destaca la necesidad de enfocar recursos y esfuerzos en áreas específicas para mantener la claridad y profundidad del estudio realizado.

1.5. Contribuciones

Las principales contribuciones de esta tesis son las siguientes:

- Un estudio sobre la viabilidad y eficiencia del uso de SYCL para el desarrollo de aplicaciones bioinformáticas en sistemas heterogéneos basados en GPUs, con el propósito de identificar las posibilidades y métodos existentes para la creación de código, así como las ventajas y desafíos relacionados en contextos de desarrollo real. Esto es especialmente relevante para desarrolladores en el ámbito de la bioinformática (y potencialmente de otras áreas) que estén considerando la migración de una aplicación existente a SYCL, o que contemplen iniciar un nuevo proyecto desde cero utilizando esta tecnología. El estudio ayudará a identificar las posibilidades y métodos para la creación de código, así como las ventajas y desafíos asociados en contextos de desarrollo real.

- La migración completa de la suite SW#, mediante la utilización de herramientas automáticas de migración de código CUDA a SYCL. SW# ofrece características avanzadas que permiten computar alineamientos tanto de secuencias de ADN como de proteínas; personalizar el algoritmo utilizado según su finalidad, como Smith-Waterman (SW), Needleman-Wunsch (NW), Semi-global (HW) y de Solapamiento (OV); ajustar el esquema de puntuación (matriz de sustitución más penalizaciones por *gaps*); entre otras. Al igual que la versión original, la versión migrada de SW# es capaz de combinar la potencia de cálculo de la CPU y la GPU, aunque en este caso se realiza a través del mismo lenguaje de programación. Al permitir configurar el número de hilos de la CPU y los dispositivos de la GPU a utilizar, se proporciona flexibilidad para diferentes configuraciones de hardware. Por último, SW# puede ser usada como una herramienta independiente tanto como una librería, lo que facilita su integración en flujos de tareas bioinformáticas más amplios. Para beneficio de la comunidad científica, la migración completa de SW# a SYCL se encuentra disponible en un repositorio web público [6].

- Un estudio exhaustivo sobre la portabilidad funcional y de rendimiento de SYCL en el ámbito de la bioinformática. Este análisis tiene en cuenta diferentes variantes de la aplicación ASB, distintas implementaciones de SYCL y su comportamiento en sistemas basados en CPU y GPU de diferentes fabricantes, así como la combinación de ambos en un entorno híbrido. Es importante destacar que no hay estudios previos que hayan utilizado un conjunto de plataformas tan amplio y diverso en la literatura existente. Este estudio es especialmente útil para investigadores y desarrolladores en el campo de la bioinformática que buscan optimizar sus aplicaciones para un procesamiento más eficiente. Al proporcionar un análisis exhaustivo sobre las variaciones en el rendimiento

[6] `https://github.com/ManuelCostanzo/swsharp_sycl`

de SYCL en distintas configuraciones de hardware, este estudio contribuye a determinar si SYCL representa una alternativa viable como modelo de programación unificado para la computación heterogénea. Además, este análisis también beneficia a la comunidad académica, proporcionando un recurso educativo para aquellos interesados en aprender sobre la portabilidad del rendimiento en diferentes entornos de cómputo y su aplicación práctica en la bioinformática.

1.6. Publicaciones

La presente tesis doctoral cuenta con el respaldo de las publicaciones que se detallan a continuación, organizadas en orden cronológico de su publicación:

- **"*Migrating CUDA to oneAPI: A Smith-Waterman Case Study*". Manuel Costanzo, Enzo Rucci, Carlos García-Sanchez, Marcelo Naiouf, Manuel Prieto-Matías. *International Work-Conference on Bioinformatics and Biomedical Engineering*. Págs 103-116. Marzo de 2022. DOI: 10.1007/978-3-031-07802-6_9. Enlace**.

  Este artículo presenta las bases fundamentales en la migración de la parte de proteínas de la suite SW# y el primer análisis de rendimientos entre CUDA y SYCL utilizando CPUs Intel y GPUs NVIDIA.

- **"*Comparing Performance and Portability between CUDA and SYCL for Protein Database Search on NVIDIA, AMD, and Intel GPUs*". Manuel Costanzo, Enzo Rucci, Carlos García-Sanchez, Marcelo Naiouf, Manuel Prieto-Matías. *2023 IEEE 35th International Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD)*. Págs. 141-148. Junio de 2023. DOI: 10.1109/SBAC-PAD59825.2023.00023. Enlace**.

  Este artículo extendió el análisis de eficiencia y portabilidad de rendimiento entre CUDA y SYCL, en un amplio y diverso conjunto de CPUs y GPUs de distintos fabricantes, incorporando también a la evaluación el uso de múltiples GPUs, tanto integradas como discretas.

- **"*Assessing Opportunities of SYCL for Biological Sequence Alignment on GPU-based Systems*". Manuel Costanzo, Enzo Rucci, Carlos García-Sanchez, Marcelo Naiouf, Manuel Prieto-Matías. *The Journal of Supercomputing*. En evaluación (R1). Enlace**.

  Este artículo cubre la migración completa de la suite de SW#, tanto para proteínas como ADN. Se realiza una comparación exhaustiva de los rendimientos a través de todas las funcionalidades de SW# en CPUs y GPUs de diferentes fabricantes. Además, se realiza una comparación entre dos implementaciones SYCL distintas, evaluando los rendimientos obtenidos en cada una.

A continuación se mencionan otras publicaciones realizadas por el tesista en relación con el tema de la tesis:

- **"*Early Experiences Migrating CUDA codes to oneAPI*". Manuel Costanzo, Enzo Rucci, Carlos García-Sanchez y Marcelo Naiouf. *Short papers of the 9th Conference on Cloud Computing Conference, Big Data & Emerging Topics*. Págs 14-18. Mayo de 2021. Enlace**.

Este artículo cubrió los primeros experimentos de migración de código CUDA a SYCL realizados por el tesista. Se seleccionaron como caso de estudio dos implementaciones CUDA originales y se evaluó la portabilidad del código y el rendimiento obtenido. Si bien no se incluye el material en esta tesis, el tema de investigación está directamente relacionado con este estudio y los conocimientos fueron aplicados en la misma.

- **"*Brief Performance Portability Analysis of a Matrix Multiplication Kernel on Multiple Vendor GPUs*". Manuel Costanzo, Enzo Rucci, Carlos García-Sanchez y Marcelo Naiouf. *Short papers of the 11th Conference on Cloud Computing Conference, Big Data & Emerging Topics*. Págs 13-18. Junio de 2023. Enlace**.

  Este artículo evalúa la portabilidad de rendimiento entre CUDA y SYCL en diferentes tipos de GPUs, seleccionando como caso de estudio la multiplicación de matrices. Si bien el material de este artículo no fue explícitamente incluido en esta tesis, los conocimientos adquiridos fueron utilizados para los posteriores artículos y para el trabajo experimental de esta investigación.

1.7. Organización de la tesis

A continuación se detalla la organización del resto del documento:

- En el Capítulo 2, se sientan las bases teóricas para los análisis y experimentos posteriores. Inicialmente, se caracterizan los sistemas heterogéneos basados en GPU. Luego, se examinan los modelos de programación para sistemas heterogéneos, incluyendo CUDA, OpenCL, OpenMP y OpenACC, analizando sus características, fortalezas y limitaciones. Posteriormente, se estudian las métricas de evaluación fundamentales para esta investigación, como el rendimiento, la portabilidad y el costo de programación. A continuación, se exploran los conceptos fundamentales de la bioinformática, centrándose en el ASB, sus diferentes usos e importancia. Por último, se analiza la paralelización de ASB y se describen las diferentes implementaciones existentes basadas en GPUs.
- En el Capítulo 3, se explica la elección de la suite SW# como caso de estudio dentro de la bioinformática. Posteriormente, se lleva a cabo la migración completa del código CUDA de SW# a SYCL utilizando la herramienta SYCLomatic. Este proceso implica la modificación del código generado, la solución de errores, la verificación funcional, la optimización y la estandarización. A continuación, se realiza una evaluación de la eficiencia de SYCLomatic como herramienta de migración y del esfuerzo de programación requerido para tal tarea. Por último, se contrastan los resultados obtenidos con trabajos relacionados.
- En el Capítulo 4, se presentan los múltiples experimentos realizados para evaluar la portabilidad funcional y de rendimiento entre las versiones CUDA y SYCL de SW#. El análisis tiene en cuenta diferentes variantes de la aplicación ASB, diversas implementaciones de SYCL y su comportamiento en sistemas basados en CPU y GPU de diferentes fabricantes, así como la combinación de ambos, en un entorno híbrido. Por último, se comparan los resultados alcanzados con los de otras investigaciones de contexto similar, resaltando las discrepancias y semejanzas entre ellos.
- En el Capítulo 5, se exponen las conclusiones de esta tesis. Asimismo, se plantean posibles trabajos futuros y áreas de investigación que podrían surgir a partir de los descubrimientos y contribuciones realizadas.

Capítulo 2

Marco Referencial

El presente capítulo se centra en el marco referencial, abarcando diversos aspectos fundamentales en el campo de los sistemas heterogéneos basados en GPUs y la bioinformática. En la Sección 2.1, se examinan los sistemas heterogéneos basados en GPUs, proporcionando una visión detallada sobre la evolución del HPC hasta la consolidación de aceleradores, y las características específicas de las GPUs, incluyendo una comparativa entre distintos fabricantes. La Sección 2.2 aborda los modelos de programación heterogénea, destacando las particularidades de plataformas como CUDA, OpenCL, OpenMP, OpenACC, Kokkos y RAJA, y su relevancia en la optimización de procesos computacionales. Posteriormente, en la sección 2.3, se presenta SYCL, una plataforma que ofrece nuevas perspectivas en la programación heterogénea. La Sección 2.4 se dedica a las métricas de evaluación, donde se analizan indicadores críticos como rendimiento, portabilidad y el costo de programación. Finalmente, la sección 2.5 se centra en la Bioinformática, explorando el ASB, clasificación de algoritmos y la implementación de estos procesos en GPUs, con un enfoque especial en la aceleración de ASB y su implementación en GPUs. Este capítulo establece una base teórica sólida, esencial para el desarrollo de la investigación y el entendimiento de las aplicaciones prácticas en el campo de estudio.

2.1. Sistemas heterogéneos basados en GPUs

En esta sección, se realiza una exploración de los sistemas heterogéneos basados en GPUs, un componente crucial en el ámbito de HPC. Se inicia con un análisis de la evolución de las arquitecturas en la Sección 2.1.1. Posteriormente, en la Sección 2.1.2, se aborda la consolidación de aceleradores, destacando su papel y evolución en sistemas computacionales modernos. La discusión se profundiza con un enfoque específico en las GPUs en la Sección 2.1.3, donde se detallan los tipos de GPUs y sus características distintivas. Finalmente, la Sección 2.1.4 proporciona un análisis comparativo de los fabricantes líderes en el mercado de GPUs, incluyendo NVIDIA, AMD e Intel.

2.1.1. HPC

Desde las primeras arquitecturas de computadoras en 1940, el objetivo fue mejorar en aspectos críticos como el costo, la energía, la seguridad y el rendimiento. Durante este pro-

ceso, se desarrollaron importantes innovaciones que transformaron el campo de la computación [43].

Un elemento esencial en esta evolución fue la interacción entre el hardware y el software, que se articula a través de las arquitecturas de conjuntos de instrucciones (ISA, por sus siglas en ingés). Las ISAs definen la forma en la que el software se comunica con el hardware subyacente y fueron fundamentales para crear oportunidades de avance en la arquitectura. Desde la apuesta de IBM en la década de 1960 por una ISA unificada en su arquitectura System/360, que revolucionó la industria y estableció el dominio de los *mainframes* IBM, hasta la microprogramación que simplificó el diseño de la unidad de control utilizando memoria en lugar de lógica cableada, estas innovaciones fueron fundamentales en la historia de la computación.

La llegada de los circuitos integrados en la década de 1970 expandió considerablemente el espacio disponible para almacenamiento de las microinstrucciones, lo que condujo al surgimiento de las computadoras con conjunto complejo de instrucciones (CISC, por sus siglas en inglés). Estos sistemas aprovecharon sus amplios almacenes de control para incorporar operaciones complejas en sus ISAs. Sin embargo, la revelación de que los compiladores utilizaban principalmente un subconjunto simple de instrucciones CISC impulsó el desarrollo de las arquitecturas con conjunto reducido de instrucciones (RISC, por sus siglas en inglés) en la década de 1980. Estas arquitecturas se enfocaron en optimizar la ejecución de operaciones simples y repetitivas, y junto con avances en compiladores y un aumento en la densidad de transistores, permitió la integración de potentes procesadores RISC en un único chip [44].

Durante el periodo anterior, el modelo tradicional de mejora de los procesadores se basó en 2 factores: en primer lugar, la explotación del paralelismo a nivel de instrucción (ILP, por sus siglas en inglés) gracias a los avances en densidad de transistores según la Ley de Moore [1]. En particular, refiere a la implementación de técnicas como la ejecución especulativa y fuera de orden, la predicción de saltos y el lanzamiento múltiple de instrucciones. En segundo lugar, el desarrollo de procesadores más rápidos gracias al aumento en la frecuencia de reloj.

El fin de la Ley de Moore y de la escalabilidad de Dennard [2] en la década de 2000 obligó a los arquitectos a buscar formas más eficientes de explotar el paralelismo, dando lugar a la era *multicore* [45]. Sin embargo, la Ley de Amdahl [3] impone límites estrictos a las mejoras en rendimiento en estos sistemas, lo que impulsó la necesidad de explorar nuevos enfoques arquitectónicos [46].

Las arquitecturas "única instrucción, múltiples datos" (SIMD, por sus siglas en inglés) ofrecen otra forma de paralelismo, donde múltiples unidades de procesamiento operan en sincronía sobre diferentes datos [47]. A pesar de ser menos flexibles que el paralelismo "múltiples instrucciones, múltiples datos" (MIMD, por sus siglas en inglés) utilizado en arquitecturas *multicore*, SIMD aprovecha de manera más eficiente los recursos para tareas con alto paralelismo de datos [48].

[1] Propuesta por Gordon Moore en 1965, esta ley predice que el número de transistores en un microchip se duplicará aproximadamente cada dos años, lo que implica un aumento continuo en el rendimiento de los procesadores.

[2] Formulada por Robert Dennard en 1974, esta ley establece que a medida que los transistores se hacen más pequeños, su consumo de energía disminuye proporcionalmente, manteniendo constante la densidad de potencia. Esta escalabilidad permitió que los chips fueran más rápidos y eficientes en energía. Sin embargo, alrededor de 2006, esta tendencia dejó de ser sostenible debido a limitaciones físicas y térmicas.

[3] Propuesta por Gene Amdahl, esta ley plantea un límite teórico al aumento de velocidad que se puede lograr en un sistema informático mediante el paralelismo. Sostiene que el rendimiento general está limitado por la porción más lenta del proceso, lo que impone restricciones significativas a las mejoras de rendimiento en sistemas con múltiples procesadores o núcleos.

2.1.2. Consolidación de aceleradores

En la actualidad, las arquitecturas específicas de dominio (DSAs, por sus siglas en inglés) representan una tendencia prometedora. Las DSAs están diseñadas y optimizadas para tareas específicas, como el procesamiento de redes neuronales e imágenes. De esta forma, aprovechan el paralelismo de manera más eficiente para el dominio objetivo, utilizan jerarquías de memoria optimizadas, emplean menor precisión numérica cuando es suficiente y pueden mapear aplicaciones eficientemente desde lenguajes de dominio específico (DSLs, por sus siglas en inglés), que exponen explícitamente el paralelismo y las estructuras de datos para un dominio en particular [49].

Arquitecturas como las GPUs, TPUs y FPGAs configurables adquirieron una importancia fundamental dentro de los sistemas HPC contemporáneos. Estos dispositivos proveen capacidad adicional para realizar cálculos en paralelo y están configurados óptimamente para trabajos específicos, lo cual puede incrementar notablemente el rendimiento en aplicaciones que se benefician de su arquitectura [50].

Entre los aceleradores más destacados, se pueden mencionar:

- GPUs: de acuerdo al listado del TOP500, es el acelerador dominante, como se puede ver en la Figura 2.1. Se trata de aceleradores especializados en el procesamiento de gráficos y operaciones matemáticas. La alta eficiencia en estas unidades se encuentra en tareas que permiten paralelización, tales como el procesamiento de imágenes y la simulación. Son altamente paralelos debido a que incorporan gran cantidad de núcleos procesadores más pequeños los cuales trabajan juntos con el fin de ejecutar operaciones rápidas y efectivas, por lo que resultan una opción óptima en el ámbito de HPC, dando lugar al concepto de "computación de propósito general en unidades de procesamiento gráfico" (GPGPU, por sus siglas en inglés) [51].

- Xeon Phi: los aceleradores Xeon Phi de Intel fueron una gama de dispositivos diseñados para HPC que empleaban una arquitectura basada en múltiples núcleos, permitiendo alcanzar un alto grado de paralelismo. Estos aceleradores se apoyaban fuertemente en la vectorización, mejorando el rendimiento de las aplicaciones. No obstante, es relevante mencionar que Intel decidió descontinuar la serie de aceleradores Xeon Phi en 2018, lo cual implica que ya no se fabrican ni se ofrecen nuevas versiones de estos dispositivos. A pesar de que los Xeon Phi resultaron altamente eficientes para aplicaciones de HPC y cargas de trabajo intensivas, su discontinuación ha impulsado la adopción de otras soluciones y tecnologías en el ámbito de la HPC [52].

- FPGA: en contraste con los aceleradores tradicionales, los FPGAs son dispositivos semiconductores flexibles en cuanto a su arquitectura porque pueden ser reconfigurados a demanda. Si bien tanto la frecuencia del reloj como el pico de rendimiento suelen ser más bajos que los correspondientes a las GPUs, la capacidad de adaptarse al problema específico a resolver le da la posibilidad a las FPGAs de obtener mejores rendimientos, además de ser en general más eficientes energéticamente [53]. No obstante, su utilidad depende en gran medida de que los recursos disponibles sean suficientes. Adicionalmente, la programación de FPGAs puede ser más desafiante en comparación con los aceleradores anteriores, por lo cual tiene grandes limitaciones de portabilidad [54].

- TPU: las unidades de procesamiento tensorial son aceleradores diseñados específicamente para el campo del aprendizaje automático. Estas unidades están optimizadas para manejar eficientemente operaciones de matrices de gran tamaño y se caracterizan por la alta eficiencia en rendimiento y consumo de energía [55]. Sin embargo, no son

versátiles, por lo que su utilidad se limita para las tareas por las que fue creado. El cómputo tensorial también está soportado en algunas GPUs y en recientes procesadores de Intel a partir de núcleos o unidades funcionales específicas/dedicadas.

Normalmente, el término "acelerador" se utiliza para referirse a un dispositivo de hardware que alivian la carga de trabajo de la CPU de manera más eficiente. Estos aceleradores pueden considerarse como coprocesadores, ya que requieren una CPU para controlar la descarga de trabajo. Al asignar tareas específicas a estos aceleradores, el sistema puede aprovechar su capacidad de procesamiento paralelo y su arquitectura optimizada para mejorar el rendimiento general de las aplicaciones. Esta interacción entre dispositivos diferentes da lugar al concepto de arquitecturas heterogéneas. La heterogeneidad permite alcanzar picos de rendimientos muchos más altos de los que se pudiera lograr solamente con la CPU, reduciendo también el consumo energético. Sin embargo, este nuevo paradigma trae consigo múltiples desafíos:

1. Programación y optimización: para lograr un mejor rendimiento es necesario comprender y aprovechar adecuadamente las características y capacidades únicas que ofrecen los diferentes tipos de aceleradores. Esto puede requerir tener un conocimiento amplio sobre la arquitectura del acelerador y las técnicas involucradas en programación paralela [56].

2. Interoperabilidad: la comunicación entre los diversos tipos de aceleradores y las CPUs convencionales puede plantear un desafío. La falta de compatibilidad entre estos puede reducir las posibilidades de contar con ciertos componentes clave en un sistema HPC. A su vez, si hay frecuentes transmisiones de información entre la CPU y el acelerador, podría haber un obstáculo para obtener un óptimo rendimiento en todo el sistema [57].

3. Elección: la elección del acelerador adecuado puede depender en gran medida de las necesidades específicas de la aplicación, siendo una decisión compleja.

2.1.3. GPUs

Como fue explicado en la Sección 2.1.2, las GPUs son dispositivos altamente paralelos creados inicialmente para el procesamiento de imágenes y videos, pero su alta capacidad de paralelismo y eficiencia energética hizo que los GPGPU se hayan vuelto populares en los últimos años. Las GPUs no pueden trabajar solas, sino que requieren del control de las CPUs para funcionar, y juntas crean un sistema heterogéneo.

2.1.3.1. Tipos de GPUs

Existen dos tipos principales de GPUs: las integradas (iGPUs, por sus siglas en inglés) y las discretas (dGPUs, por sus siglas en inglés). La Figura 2.2 muestra gráficamente la diferencia entre sistemas con ambos tipos. Por un lado, las iGPUs son comunes en procesadores de propósito general, como las unidades de procesamiento acelerado (APUs, por sus siglas en inglés) y los sistemas en chip (SoCs, por sus siglas en inglés). Estas GPUs comparten recursos, como la memoria y el ancho de banda, con la CPU principal. A pesar de tener menos núcleos de procesamiento en comparación con las dGPUs, presentan beneficios en cuanto a eficiencia energética y tamaño compacto. Además, su mayor ventaja radica en la eliminación del costo de transferencia de datos a través del bus de comunicación PCIe, el cual es lento y afecta considerablemente el rendimiento. El objetivo de las iGPU se centra en equilibrar el

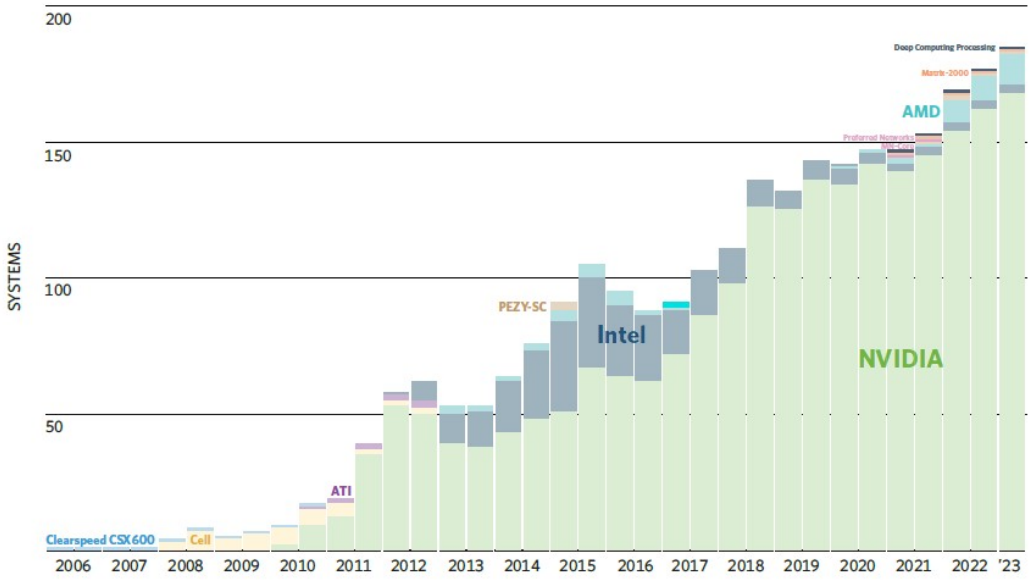

Figura 2.1: Cantidad de sistemas que utilizan aceleradores del ranking TOP500 hasta el 2023 (extraído de [58]).

rendimiento y la eficiencia para una amplia gama de aplicaciones de computación cotidianas, como la reproducción de video y el procesamiento básico de gráficos.

Por otro lado, las dGPUs son dispositivos independientes que se conectan a través de una ranura de expansión en la placa madre. Estas GPUs están diseñadas específicamente para cómputo intensivo y se caracterizan por tener una mayor cantidad de núcleos de procesamiento, lo que les permite ofrecer un mayor rendimiento en aplicaciones HPC. Además, las dGPUs suelen disponer de su propia memoria dedicada, mejorando la eficiencia y la capacidad para manejar volúmenes de datos más grandes.

2.1.3.2. Características

A continuación, se describen algunas de las principales características del modelo de ejecución de las GPUs:

Arquitectura paralela: la GPUs se caracterizan por su diseño altamente paralelo, estructurado alrededor de una matriz de núcleos de ejecución que permiten el procesamiento simultáneo de miles de unidades (a modo de ejemplo, las últimas GPUs NVIDIA tienen más de 16 mil unidades de procesamiento paralelo). Esta capacidad se basa en el modelo *Single Instruction Multiples Threads* (SIMT), un concepto similar a SIMD, que resulta ser una estrategia especialmente efectiva para tareas que involucran operaciones matriciales y vectoriales, fundamentales en aplicaciones gráficas y en el campo de la computación científica [59].

Las GPUs se diferencian de las CPUs principalmente por su capacidad para manejar eficientemente operaciones de alta latencia, como los accesos a la memoria principal, sin necesidad de cachés extensos. Este logro se basa en su habilidad para manejar múltiples hilos de ejecución simultáneamente y en su enfoque en el *multithreading* de grano fino, lo que

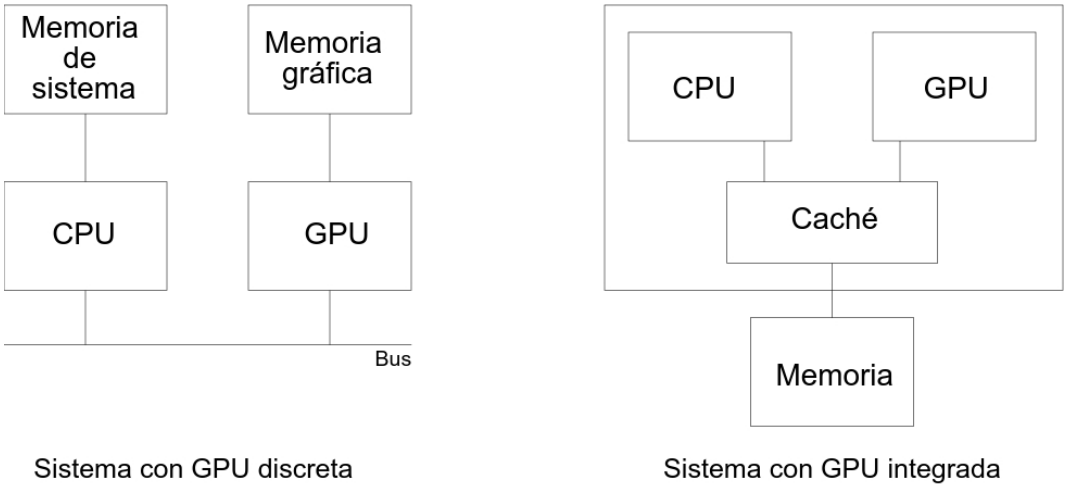

Figura 2.2: Esquemas de sistemas con dGPU (izquierda) e iGPU (derecha).

les permite alternar rápidamente entre hilos en cada ciclo de reloj. Este diseño maximiza el paralelismo a nivel de instrucción y aprovecha una amplia gama de registros para almacenar el contenido de instrucciones de múltiples hilos, minimizando así el impacto de los tiempos de acceso a memoria. Por otro lado, las CPUs, con una lógica de control más compleja y una menor densidad de cálculo, están optimizadas para secuencias de instrucciones con baja latencia y tareas que requieren altas velocidades de reloj y ejecución rápida de instrucciones individuales. A diferencia de las GPUs, que tienen *pipelines* más profundos (con cientos de etapas) para manejar alta latencia y procesar mayor volumen de datos, las CPUs tienen *pipelines* menos profundos (con menos de 30 etapas), lo que resulta en una baja tolerancia a la latencia. Además, las GPUs modernas comenzaron a incorporar lógicas de control de flujo más avanzadas y técnicas de acceso a memoria *scatter/gather* [4], acercándose en ciertos aspectos a las CPUs. La Figura 2.3 ilustra esta comparación.

Jerarquía de Memoria: presenta una jerarquía de memoria profunda y compleja que habitualmente se compone de 4 niveles:

- Memoria global: es accesible por todos los hilos, pero con alta latencia, por lo que es importante minimizar el acceso a esta memoria para optimizar el rendimiento.
- Memoria compartida: es más rápida que la memoria global y es accesible solo por hilos dentro del mismo bloque, lo que resulta útil para compartir datos entre hilos.
- Memoria constante: es una porción de solo lectura de la memoria global y cuenta con una caché especial que permite realizar accesos de lectura más rápidos.
- Memoria privada: es privada para cada hilo y se utiliza para almacenar datos temporales, por lo cual su tamaño es pequeño.

[4]La técnica *scatter/gather* es un método de manejo de memoria que permite a un procesador leer o escribir datos en múltiples y no consecutivas ubicaciones de memoria en una sola operación, mejorando la eficiencia en el procesamiento de datos.

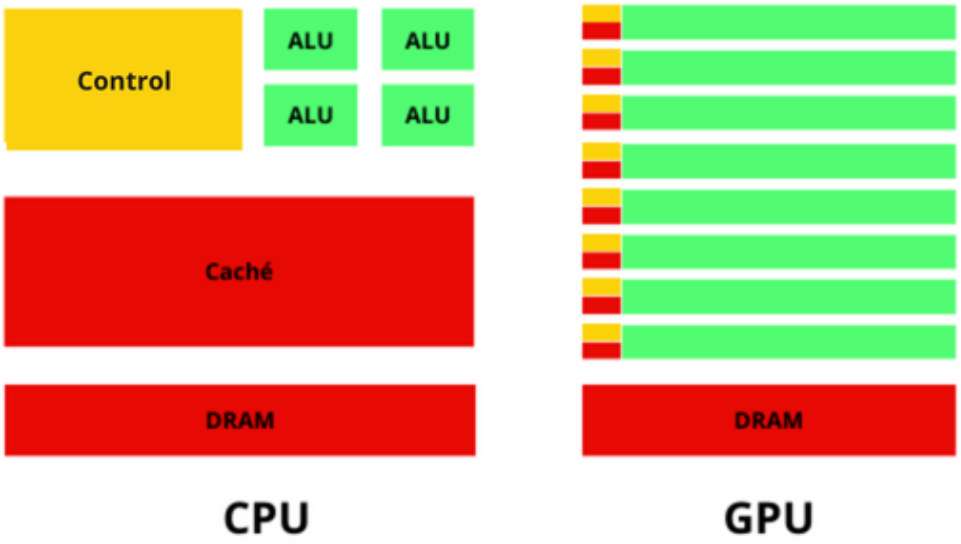

Figura 2.3: Comparación de arquitecturas entre CPU y GPU.

Uso Eficiente de la Memoria: existen 2 mecanismos:

- Coalescencia de Memoria: agrupa solicitudes de memoria de hilos cercanos para reducir el número de transacciones de memoria.
- *Prefetching* y *Caching*: son técnicas para buscar y almacenar datos en caché, reduciendo la latencia de acceso a la memoria.

Gestión de latencia: es importante reducir la latencia en las GPUs, ya que si un hilo está esperando una operación de memoria, la GPU puede cambiar a otro hilo listo para ejecutar, maximizando la utilización de los recursos de cómputo.

Programación Paralela:

- Organización de Hilos: las GPUs estructuran los hilos en agrupaciones coordinadas para maximizar la ejecución paralela, aumentando significativamente la eficiencia del procesamiento. El tamaño y la configuración de estos grupos de hilos varían según el modelo específico y el fabricante de la GPU, permitiendo una flexibilidad adaptativa a diferentes tipos de cargas de trabajo y aplicaciones.
- Unidades de Procesamiento Paralelo: estas unidades son responsables de ejecutar simultáneamente múltiples conjuntos de hilos, lo cual facilita el paralelismo masivo en las GPUs.

.

Alta Capacidad de Cálculo: las GPUs modernas ofrecen *teraflops* de rendimiento y cientos de *gigabytes* por segundo de ancho de banda de memoria, lo que les permite manejar grandes conjuntos de datos y realizar cálculos complejos a alta velocidad. Sin embargo, alcanzar estos niveles de rendimiento puede ser desafiante y requiere un diseño cuidadoso y una comprensión profunda de la arquitectura de la GPU [60].

2.1.4. Fabricantes

Tres compañías destacadas en la industria de las GPUs son NVIDIA, AMD e Intel. Durante el segundo trimestre del año 2023, NVIDIA se posiciona como líder en el mercado de dGPUs con una participación del 87 %, mientras que AMD e Intel ostentan el 10 % y el 3 %, respectivamente. En cuanto a las iGPUs, Intel domina con un 84 % del mercado, mientras que AMD cuenta con el 16 % [8]. Ambas estadísticas se reflejan en la Figura 2.4.

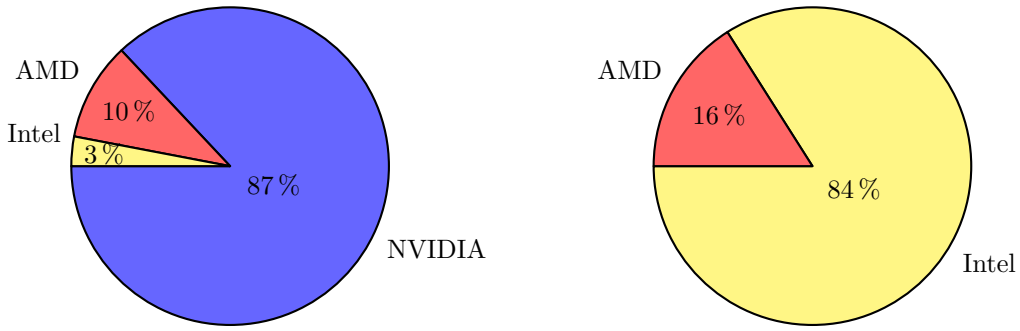

Figura 2.4: Distribución de mercado de dGPUs (izquierda) e iGPUs (derecha).

2.1.4.1. GPUs de NVIDIA

Las GPUs NVIDIA son dispositivos diseñados para procesar gráficos y acelerar el rendimiento de aplicaciones que requieren una gran cantidad de cómputo en paralelo. A continuación, se presentan algunas de las características principales [61]:

- Arquitectura CUDA: es una arquitectura de procesamiento paralelo que permite a los desarrolladores programar la GPU para realizar cálculos complejos de manera más eficiente [62]. En la Figura 2.5 se ilustra un ejemplo.
- Núcleos CUDA: son los procesadores de la GPU que realizan el cómputo. Las GPUs NVIDIA tienen una gran cantidad de núcleos CUDA, lo que les permite procesar grandes cantidades de datos de manera simultánea [63].
- *Streaming Multiprocessors* (SM): Los SM son el corazón de la arquitectura CUDA de NVIDIA. Cada SM contiene un conjunto de núcleos CUDA y es responsable de ejecutar múltiples hilos de instrucciones en paralelo. Esta capacidad permite a las GPUs NVIDIA manejar eficientemente tareas complejas y cálculos intensivos. La eficiencia y el rendimiento de un SM son fundamentales para determinar el rendimiento general de la GPU. Los avances en la tecnología SM impulsaron mejoras significativas en el procesamiento paralelo.
- Memoria de mayor ancho de banda: las GPUs de NVIDIA cuentan con una memoria de alto ancho de banda, lo que les permite manejar grandes volúmenes de datos a altas velocidades. Esta capacidad resulta fundamental para aplicaciones que requieren un procesamiento intensivo y rápido de datos. En particular, algunas GPUs de NVIDIA incorporan la tecnología de memoria avanzada denominada *High Bandwidth Memory* (HBM), la cual fue especialmente diseñada para ofrecer un ancho de banda significativamente mayor en comparación con las memorias tradicionales. Este tipo de memoria

resulta esencial para lograr un rendimiento óptimo en tareas de computación gráfica y procesamiento de datos a gran escala.

- Tecnología de trazado de rayos: es una innovación que posibilita a las GPUs NVIDIA simular de manera precisa la interacción de la luz con los objetos en una escena, logrando así la creación de gráficos de mayor realismo. Esta tecnología se encuentra disponible desde la serie Volta.
- Tensor Cores: son núcleos especializados que se utilizan para acelerar el procesamiento de redes neuronales y otros tipos de aprendizaje automático. Esta tecnología se encuentra disponible desde la serie Turing.
- Conexión con la CPU: las GPUs generalmente se conectan a la CPU a través del bus PCI, proporcionando una interfaz estándar para la comunicación entre la GPU y el resto del sistema. Sin embargo, algunas GPUs de alta gama utilizan tecnologías avanzadas como NVLINK, que ofrece una mayor velocidad de transferencia de datos y ancho de banda que el bus PCI tradicional, mejorando así la eficiencia y el rendimiento en aplicaciones de cómputo intensivo.

Las GPUs de NVIDIA han experimentado avances a lo largo del tiempo, desde su primer modelo para HPC (Tesla) hasta el más reciente (Ada Lovelace) [64]:

1. Tesla (2007): marcó el comienzo de la era GPGPU con la introducción de la arquitectura unificada de *shaders* y la tecnología CUDA. Esta tecnología permitió a los desarrolladores utilizar GPUs para tareas de cómputo no gráficas, ampliando significativamente su aplicabilidad en diversos campos como la simulación científica, la criptografía y la investigación biomédica. La arquitectura Tesla también introdujo importantes mejoras en el rendimiento de cálculo y la eficiencia energética, estableciendo un nuevo estándar en la HPC. [65].
2. Fermi (2010): introdujo innovaciones clave para GPGPUs, como la memoria caché L2 unificada y corrección de errores de memoria, esenciales para aplicaciones de HPC [66].
3. Kepler (2012): mejoró significativamente la eficiencia energética, un factor crucial en contextos de cómputo intensivo. Su tecnología *GPU Boost* [5] optimizó el rendimiento de las GPUs, mientras que el *Dynamic Parallelism* [6] permitió a las aplicaciones adaptarse dinámicamente a la carga de trabajo de la GPU, facilitando la programación paralela [67].
4. Maxwell (2014): aplicó mejoras en la eficiencia energética y el rendimiento en el cómputo de GPGPU, optimizando tareas de procesamiento paralelo. Maxwell avanzó en tecnologías relevantes para HPC, como mejoras en la arquitectura de memoria, incluyendo la compresión de memoria dinámica, una caché L2 aumentada, memoria unificada, y en la eficiencia de cálculo.
5. Pascal (2016): representó un salto en rendimiento para GPGPU, con la introducción de la tecnología *CUDA Compute Capability 6.0*, que mejoró significativamente el procesamiento paralelo y la eficiencia energética. También incorporó la interconexión de alta velocidad *NVLink* y soporte mejorado para aplicaciones de aprendizaje profundo y HPC.

[5] La tecnología *GPU Boost* permite aumentar automáticamente la velocidad del reloj de la GPU.

[6] *Dynamic Parallelism* es una tecnología que permite a los *kernels* de la GPU lanzar otros *kernels* directamente.

6. Volta (2017): se enfocó en la inteligencia artificial (IA) y el aprendizaje profundo incorporando los núcleos tensoriales, optimizando las operaciones de matriz fundamentales en estas áreas. También mejoró significativamente el rendimiento en cómputo de precisión mixta, crucial para simulaciones científicas y análisis de grandes volúmenes de datos.

7. Turing (2018): introdujo el trazado de rayos en tiempo real y mejoró el procesamiento de IA. La tecnología *Adaptive Shading* optimizó la calidad de imagen y redujo la carga de trabajo en la GPU, beneficiando aplicaciones de gráficos avanzados y juegos.

8. Ampere (2020): avanzó en el rendimiento de trazado de rayos y operaciones de IA con la memoria GDDR6X. Los Tensor Cores de tercera generación mejoraron aún más el rendimiento.

9. Lovelace (2022): priorizó el paralelismo masivo con más unidades de núcleos CUDA, esencial para tareas de cómputo intensivo y simulaciones complejas. Las mejoras en el trazado de rayos y en la tecnología de hardware de segunda generación impulsaron aún más la calidad de gráficos e IA [68].

Figura 2.5: Diagrama representativo de la arquitectura CUDA (Ampere).

2.1.4.2. GPUs de AMD

Las primeras GPUs AMD surgieron en el año 1986, cuando ATI Technologies Inc. ingresó al mercado de las tarjetas gráficas. Durante un tiempo, ATI se mantuvo como una empresa independiente, pero en 2006, AMD adquirió la compañía, lo que marcó un punto inicial hacia la innovación en el campo de las GPUs [69]. La Figura 2.6 representa gráficamente la arquitectura de las dGPUs AMD.

A continuación, se presentan algunas de las características más destacadas de estas GPUs:

- **Unidades de Cómputo (CU, por sus siglas en inglés)**: representan el equivalente a los SMs de las GPUs de NVIDIA. Cada CU contiene unidades de procesamiento Aritmético-Lógicas (ALUs, por sus siglas en inglés) y unidades de texturizado. Además, tiene la capacidad de ejecutar varios hilos simultáneamente, lo que contribuye a un alto rendimiento.

- **Sistema de caché**: incorporan cachés L1 y L2 que contribuyen a agilizar el acceso a datos y texturas, disminuyendo la latencia y optimizando el rendimiento en procesamiento gráfico y de cómputo en general.
- **Controladores de memoria**: poseen controladores de memoria y *buses* especializados que se enlazan con la memoria GDDR dedicada, lo que permite un alto ancho de banda para el procesamiento eficiente de datos gráficos.
- **Soporte para APIs**: brindan un extenso respaldo a las APIs de gráficos y computación paralela, como DirectX, OpenGL y Vulkan para gráficos, y OpenCL para computación paralela. De la misma forma que CUDA es una tecnología exclusiva de NVIDIA, AMD cuenta con su equivalente, denominado Radeon Open Compute (ROCm).

Al igual que sucede con NVIDIA, las GPUs de AMD han evolucionado con el paso del tiempo. En esta ocasión, se pueden clasificar en dos categorías: integradas y discretas.

Por un lado, las iGPUs DE AMD fueron evolucionando de la siguiente forma:

- Radeon HD Series (2011): esta serie fue una de las primeras en ofrecer gráficos integrados de alto rendimiento en las APU de AMD. Introdujo varias mejoras en la eficiencia energética y en el rendimiento por vatio. La arquitectura HD proporcionó un soporte decente para juegos y aplicaciones gráficas en ese momento, incluyendo soporte para *DirectX 11*. También ofreció una mejor integración con la CPU, lo que permitió un procesamiento más eficiente de tareas gráficas y de cómputo en paralelo [70].
- Radeon R Series (2014): esta serie incluye mejoras en la eficiencia energética y el rendimiento gráfico en comparación con las generaciones anteriores. Utiliza la arquitectura *Graphics Core Next* (GCN), que proporciona un mejor equilibrio entre el procesamiento de gráficos y las aplicaciones de computación. La serie R también incluye soporte para *DirectX 12*, lo que permite una mejora en los juegos y aplicaciones gráficas. Esta arquitectura es conocida por su capacidad para ofrecer un rendimiento gráfico aceptable en un segmento integrado [71].
- Radeon Vega (2017): esta arquitectura representa un cambio significativo respecto a las generaciones anteriores. Introduce una nueva organización en CU, que mejora el paralelismo y la eficiencia energética. Vega utiliza la tecnología HBM2, que ofrece un ancho de banda mayor comparado con GDDR5, lo que resulta en un rendimiento superior para tareas gráficas y de computación en general. Además, Vega integra un motor de procesamiento de video mejorado y tecnología de trazado de rayos en tiempo real para gráficos más realistas [72].

Por otro lado, las dGPUs AMD también fueron evolucionando:

- GCN (2012): la primera versión de la arquitectura GCN se lanzó en 2012 y se caracterizó por el énfasis hacia el modelo GPGPU. Desde la perspectiva del hardware, representó un cambio de filosofía de procesadores *Very Long Instruction Word* a SIMD. En esa arquitectura, cada CU contiene 4 unidades vectoriales de 16 carriles cada una, lo que le permite tener 16 hilos en ejecución en un momento dado. Además, GCN mejoró notablemente la productividad en la ejecución de instrucciones, lo que se tradujo en un aumento en las instrucciones por ciclo (IPC). Desde su primera versión, GCN fue evolucionado a través de diferentes versiones, cada una destacando por mejoras en aspectos como el rendimiento del sistema de caché, optimizaciones en la gestión de

energía y avances en el soporte de tecnologías de renderizado. Cada versión incorpora mejoras en las memorias de caché L0, L1 y L2, reflejando un enfoque constante en la eficiencia y el rendimiento. La última versión es la 5.1, presentada en 2020 [73].

- RDNA (2019): esta arquitectura introdujo cambios significativos en comparación con la serie anterior. Se enfocó en mejorar la eficiencia energética y el rendimiento por ciclo de reloj mediante la implementación de una nueva estructura de UC que mejoró notablemente el IPC. También optimizó el acceso a las memorias caché L1 y L2 y mejoró la eficiencia energética de estos dispositivos, ubicándose como una buena opción para HPC. [74].

- RDNA 2 (2020): representa un avance significativo en comparación con RDNA, ya que incorpora un concepto denominado "Caché Infinito", que actúa como una *caché de víctimas*, recolectando aquellas líneas que son descartadas por la caché L2. Esto logra una mayor eficiencia en la recuperación de datos y se reduce el consumo energético en ciertas operaciones [75].

- RDNA 3 (2023): cuenta con aceleradores de IA innovadores y aceleradores de trazado de rayos de segunda generación, lo que permite una aceleración de hasta 2.7×. Además, esta arquitectura presenta un diseño *chiplet* avanzado, similar al utilizado en las CPUs de AMD, y ofrece un rendimiento por vatio consumido un 54 % superior al de RDNA 2. También se incorporan mejoras como una interconexión de *chiplets* ultrarrápida, una memoria ampliada, un bus de memoria más ancho y el nuevo *AMD Radiance Display Engine*, que mejora la grabación y transmisión de video [76].

2.1.4.3. GPUs de Intel

En la década de 1990, Intel empezó a desarrollar sus primeras iGPU, que se ubicaban en la placa madre, como parte de la arquitectura *Hub* de Intel. En el año 2010, Intel presentó la primera generación de su tecnología *HD Graphics*, renombrada en 2017 como *Intel UHD Graphics*. Con esta tecnología, se inició la integración de las GPUs directamente en el mismo chip que la CPU, como parte del *Platform Controller Hub* de Intel.

Como se explicó en la Sección 2.1.3.1, las iGPUs y dGPUs presentan diferencias significativas en términos de ubicación y rendimiento. En la Figura 2.7 y la Figura 2.8 se presentan las arquitecturas de las iGPUs y dGPUs de Intel, respectivamente. Las iGPUs comparten la misma memoria RAM con la CPU, lo que implica que todas las tareas del sistema, incluyendo las gráficas, se ejecutan utilizando la misma memoria [78]. Aunque las iGPU de Intel cuentan con una pequeña cantidad de memoria integrada, su rendimiento se ve afectado por la velocidad y cantidad de la RAM del sistema [79]. Por otro lado, las dGPUs, tienen su propia memoria RAM independiente que es exclusivamente utilizada por la GPU, lo que resulta en un rendimiento considerablemente mejor que las iGPU.

Con el paso del tiempo, Intel ha evolucionado en el desarrollo de sus GPUs. Por un lado, esta empresa desarrolló diferentes tipos de iGPUs:

- Intel Extreme Graphics (2002): esta GPU trajo mejoras en el procesamiento tridimensional y soporte para *DirectX 7*. Tenía 64 MB de memoria de video y podía generar gráficos 3D de alta calidad. Esta GPU representó un avance significativo en las capacidades gráficas integradas de Intel, aunque aún se encontraba limitada en comparación con las soluciones especializadas disponibles en ese momento.

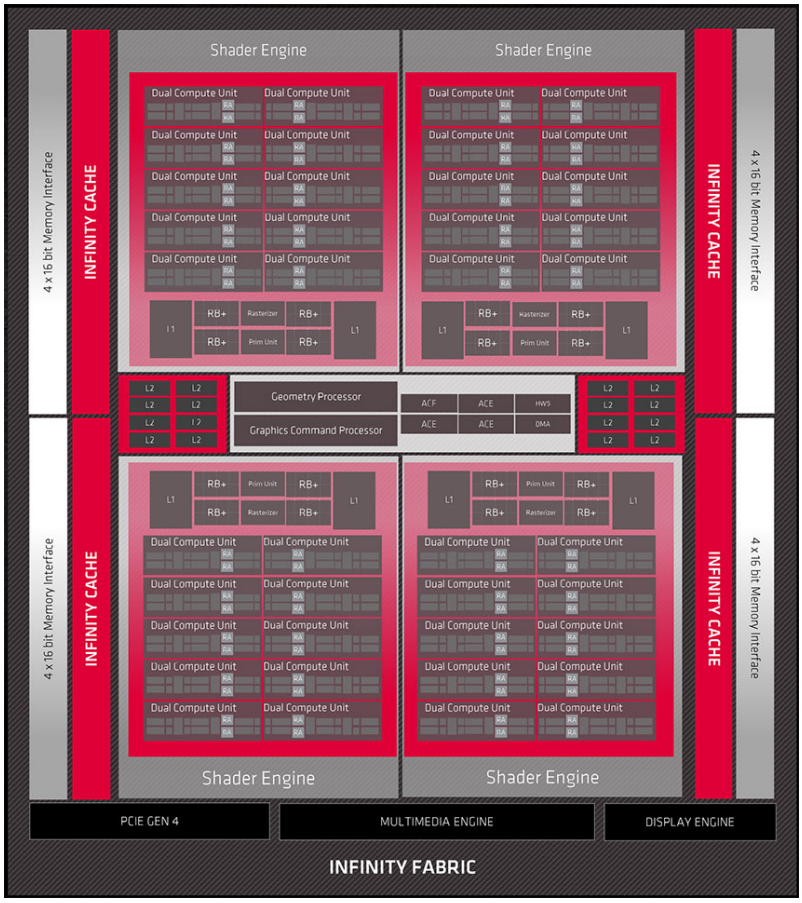

Figura 2.6: Diagrama representativo de la arquitectura dGPU de AMD (RDNA 2) (extraído de [77]).

- Intel Graphics Media Accelerator (2004): se implementaron mejoras en el rendimiento de video y 3D, lo cual tuvo un impacto significativo en el progreso hacia una mayor eficiencia energética.
- Intel HD Graphics (2010): representó un avance importante en el rendimiento gráfico integrado, al brindar un mejor soporte para *DirectX 10* y *OpenGL 3.1*. Tenía 128 MB de memoria de video y podía generar gráficos 3D de alta calidad. Proporcionó una alternativa más adecuada para aplicaciones de computación gráfica de nivel moderadamente exigente.
- Intel Iris Graphics/Iris Pro (2013): estas GPUs presentaron un rendimiento significativamente superior en comparación con Intel HD Graphics, debido a una mayor cantidad de unidades de ejecución y una caché eDRAM en ciertos modelos. La HD Graphics tenía 128 MB de memoria de video, mientras que la Iris Graphics/Iris Pro tenía 1 GB o 2 GB de memoria de video.
- Intel UHD Graphics (2017): esta GPU se centró en optimizar el consumo de energía y el rendimiento en aplicaciones 4K. La UHD Graphics fue una GPU popular para los

juegos y aplicaciones de gráficos 3D avanzados.

- Intel Iris Plus Graphics (2017): mejoró aún más el rendimiento con más unidades de ejecución y frecuencias más altas, lo que resultó en un incremento en el rendimiento para la visualización de datos y aplicaciones gráficas en HPC.
- Intel Iris Xe Graphics (2020): representa un avance significativo en las iGPUs, ya que cuenta con el respaldo de características modernas como *Intel Deep Learning Boost* y mejoras notables en el rendimiento de cálculo y eficiencia, lo cual resulta fundamental en entornos de HPC. Tiene 4 GB u 8 GB de memoria de video, y puede generar gráficos 3D de calidad ultra alta.

Por otro lado, Intel se fue insertando en el mercado de dGPUs:

- Intel740 (1998): esta GPU fue una de las pioneras en incorporar aceleración 3D en el mercado masivo. Tenía 128 KB de memoria de video y podía generar gráficos 3D acelerados en tiempo real.
- Larrabee (2009): predecesor directo de la Intel Xeon Phi, buscaba introducir una arquitectura de GPU basada en una red de núcleos x86. Aunque no llegó a ser lanzado comercialmente como una GPU, sus conceptos arquitectónicos tuvieron un impacto significativo en el desarrollo posterior de tecnologías de cómputo paralelo y gráficos en Intel. Su enfoque en la programación más familiar para los desarrolladores y su potencial para la computación paralela a gran escala lo convertían en una opción prometedora para HPC [80].
- Intel Iris Xe Max (2020): esta fue la primera dGPU de Intel enfocada principalmente en laptops y dispositivos móviles. Aunque no está especialmente diseñada para HPC, cuenta con características como la aceleración de codificación y decodificación de video por hardware, así como la tecnología *Deep Link* de Intel, que puede mejorar la eficiencia y el rendimiento en ciertas aplicaciones de HPC al trabajar en conjunto con la CPU.
- Intel Arc Alchemist (2022): Intel se inserta en el mercado de GPU de alto rendimiento con esta serie, la cual cuenta con una arquitectura más avanzada y ofrece soporte para trazado de rayos e IA. Arc Alchemist también presenta un potencial prometedor para aplicaciones HPC, especialmente aquellas que se benefician del trazado de rayos y el aprendizaje automático, como la simulación científica y la visualización de datos [81].

2.2. Modelos de programación heterogénea

La computación heterogénea busca alcanzar la eficiencia y el rendimiento óptimo mediante la explotación de los diferentes recursos de hardware disponibles. En esta sección se presentan seis modelos de programación clave que facilitan el desarrollo de software en entornos heterogéneos: CUDA, OpenCL, OpenMP, OpenACC, Kokkos y RAJA. Cada uno de estos modelos ofrece un enfoque distinto para la programación paralela, lo que permite a los desarrolladores maximizar el rendimiento al utilizar distintos tipos de procesadores, como CPUs, GPUs y otros aceleradores.

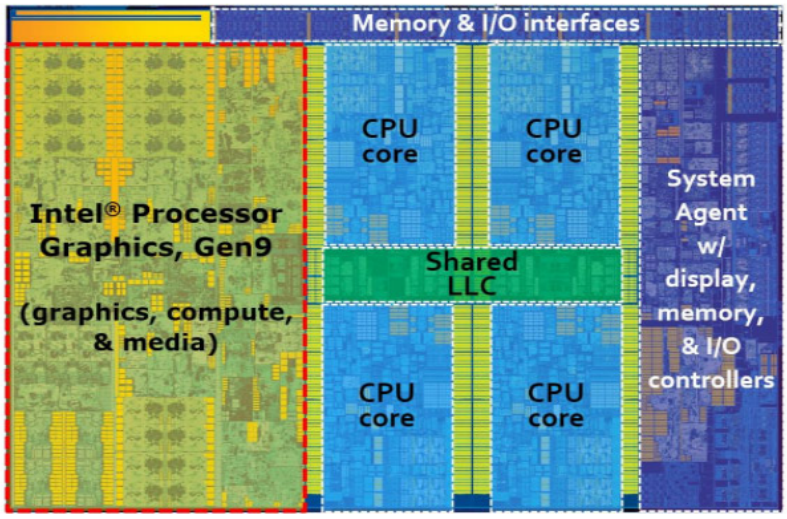

Figura 2.7: Diagrama representativo de la arquitectura iGPU de Intel (Intel UHD Graphics) (extraído de [82]).

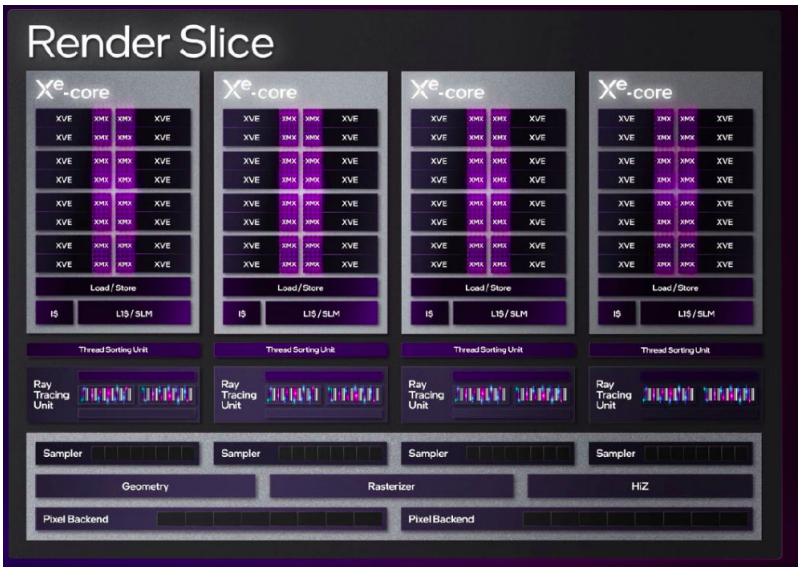

Figura 2.8: Diagrama representativo de la arquitectura dGPU de Intel (Xe Max) (extraído de [83]).

2.2.1. CUDA

CUDA es una plataforma y un modelo de programación presentado por primera vez en 2007 por NVIDIA. La Figura 2.9 ilustra un diagrama del modelo de ejecución, de memoria y de plataforma de CUDA. El objetivo es aprovechar la potencia de las GPUs para realizar cómputo general, en lugar de limitarse exclusivamente a tareas relacionadas con gráficos, capacidad que permitió a los desarrolladores utilizar las GPUs para una amplia variedad de

tareas computacionales [84].

2.2.1.1. Modelo de plataforma

El enfoque de programación de CUDA se basa en el modelo *host-device*. El código que se ejecuta en la CPU se conoce como código del *host*, mientras que el código que se ejecuta en la GPU se conoce como código del dispositivo (*device*).

El *host* se encarga de la gestión de tareas, como la asignación de recursos y la transferencia de datos entre la CPU y la GPU, mientras que el dispositivo se encarga de realizar los cálculos en paralelo en la GPU, a través de funciones llamadas *kernels*. Estos *kernels* se invocan desde el código principal de la CPU y son ejecutados por miles de hilos primitivos desde la GPU.

Además de los *kernels*, CUDA ofrece un conjunto de librerías y herramientas que facilitan el desarrollo de aplicaciones paralelas en la GPU, tales como funciones optimizadas para operaciones matemáticas, procesamiento de imágenes, álgebra lineal, entre otros.

2.2.1.2. Modelo de ejecución

En el núcleo del modelo de ejecución de CUDA se encuentra el concepto de hilo, que es una secuencia de instrucciones que se ejecuta de manera independiente en una GPU. Estos hilos se agrupan en bloques, y los bloques a su vez forman un *grid*. Cada uno de estos elementos tienen un identificador único que permite a los programadores controlar y coordinar la ejecución paralela.

Este modelo de ejecución se basa en el principio de dividir el trabajo en tareas más pequeñas y manejables, donde cada hilo se encarga de realizar una parte del trabajo, y los mismos se organizan en bloques para aprovechar al máximo la capacidad de procesamiento de la GPU. A medida que los bloques se ejecutan en la GPU, CUDA se encarga de asignar los hilos a los recursos de cómputo disponibles, como los SM y los *kernels* de la GPU.

En este contexto, un componente fundamental es el concepto de *warp*. Un *warp* consiste en 32 hilos que se ejecutan de manera conjunta. Esta agrupación es importante debido a que dentro de un SM, hay una unidad de *scoreboarding* [7] que administra la ejecución, pero lo hace a nivel de *warp*. Este enfoque permite una gestión más eficiente de los recursos de la GPU, optimizando el rendimiento y la sincronización de los hilos dentro de cada *warp* [85].

Además de la asignación de hilos y la gestión de la memoria, el modelo de ejecución de CUDA también proporciona mecanismos para la sincronización y la comunicación entre los hilos. Los programadores pueden utilizar instrucciones especiales, como barreras de sincronización, para asegurarse de que los hilos se coordinen adecuadamente y eviten condiciones de carrera o resultados inconsistentes.

En la Figura 2.10 se ilustra un ejemplo de suma de vectores en CUDA.

2.2.1.3. Modelo de memoria

El modelo de memoria de CUDA cuenta con diferentes niveles que se organizan jerárquicamente: la memoria global, la memoria compartida, la memoria constante y la memoria local:

- Memoria global: es accesible por todos los hilos y se utiliza para almacenar datos que deben ser compartidos entre ellos. Los datos en la memoria global son persistentes y se mantienen incluso después de que finaliza la ejecución de un kernel.

[7] Scoreboarding es una técnica de control de hardware utilizada en procesadores para gestionar la ejecución de instrucciones, manteniendo un seguimiento y resolviendo conflictos de recursos para evitar que se produzcan cuellos de botella y garantizar un uso eficiente de la unidad de procesamiento.

- Memoria compartida: es una memoria de baja latencia que se comparte entre los hilos de un bloque. Se utiliza para la comunicación y la sincronización entre los hilos y los datos almacenados en la memoria compartida son rápidamente accesibles por ellos dentro del mismo bloque.
- Memoria constante: se utiliza para almacenar datos que no cambian durante la ejecución del *kernel*. La memoria constante ofrece una latencia de acceso baja y una alta tasa de transferencia de datos. Es útil para almacenar constantes y tablas de búsqueda que se utilizan en el cálculo.
- Memoria local: se refiere a la memoria privada asignada a cada hilo. Cada uno tiene su propia memoria local para almacenar variables y resultados temporales. La memoria local es de acceso rápido, pero su tamaño es limitado. Si se produce un desbordamiento de memoria local, los datos se almacenan en la memoria global, lo que puede afectar el rendimiento.

Además de estos tipos de memoria, CUDA también proporciona registros, que son memoria local de alta velocidad dentro de cada SM y se utilizan para almacenar variables y resultados temporales de los hilos.

2.2.2. OpenCL

OpenCL es un estándar de programación de código abierto, desarrollado en 2008 por el Grupo Khronos, que permite la programación paralela de diversos dispositivos, como CPUs, GPUs, FPGAs, entre otros, mediante un único código fuente. OpenCL es un marco de trabajo que incluye una API, librerías y un sistema de *runtime* para respaldar el desarrollo de software. Mediante OpenCL, los programadores pueden escribir programas de propósito general que se ejecutan en aceleradores sin necesidad de implementar sus algoritmos en un lenguaje diferente para cada uno. Es por esto que el objetivo principal de OpenCL es permitir a los programadores escribir código portable y eficiente abstrayendo el hardware subyacente [11].

2.2.2.1. Modelo de plataforma

OpenCL define una plataforma como la combinación de un conjunto de dispositivos de cómputo y los controladores correspondientes que permiten la ejecución de *kernels* en dichos dispositivos.

Una plataforma OpenCL puede consistir en múltiples dispositivos de cómputo, como CPUs y GPUs de diferentes fabricantes. Cada dispositivo dentro de la plataforma tiene su propio controlador que proporciona una interfaz común para la ejecución de *kernels* y la gestión de la memoria.

El modelo de plataforma de OpenCL permite que las aplicaciones desarrolladas en este sean portables entre diferentes sistemas y dispositivos, lo que significa que una aplicación OpenCL puede ejecutarse en diferentes plataformas sin necesidad de realizar modificaciones significativas en el código fuente. Esto se logra mediante la definición de una API estándar que proporciona funciones y estructuras de datos que son independientes de la arquitectura específica del dispositivo [86].

Además de la portabilidad, OpenCL también promueve la interoperabilidad entre diferentes lenguajes y entornos de programación como C, C++, Python y Java, lo que permite combinar fácilmente el código OpenCL con código escrito en otros lenguajes.

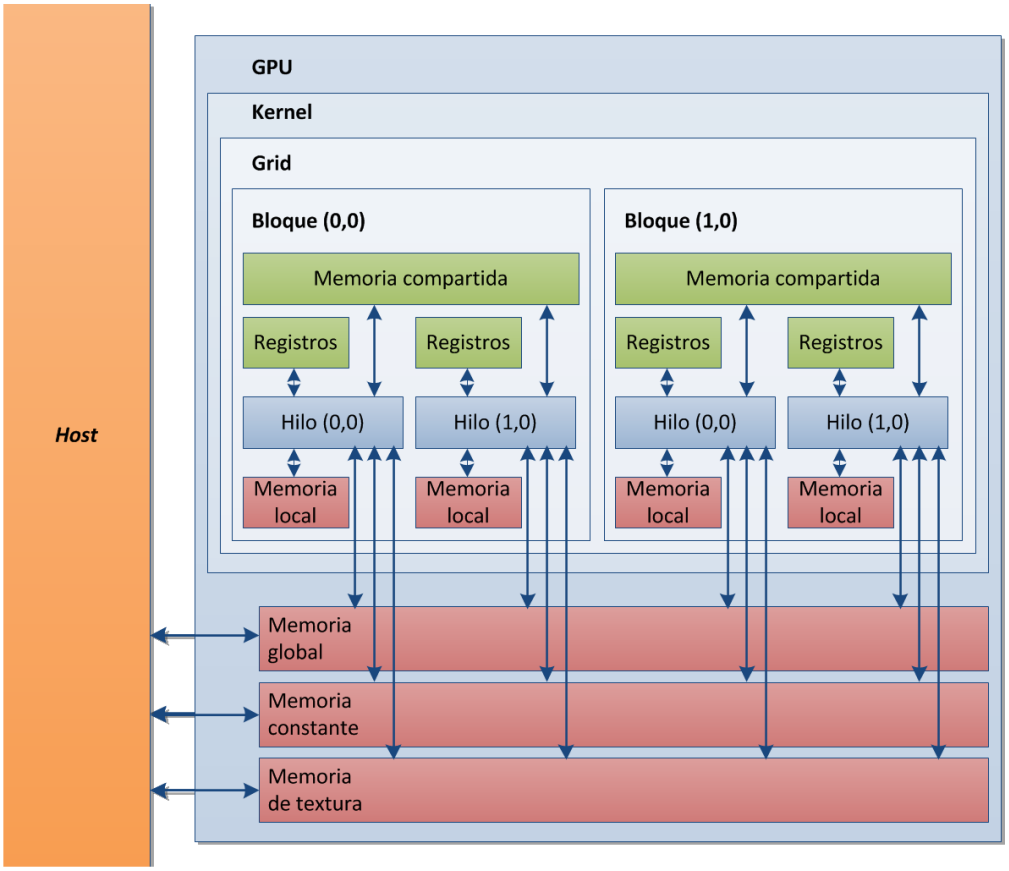

Figura 2.9: Diagrama del modelo de ejecución, de plataforma y de memoria de CUDA (extraído de [6]).

2.2.2.2. Modelo de ejecución

En OpenCL, una aplicación se divide en un conjunto de tareas independientes conocidas como *kernels*. Cada uno de estos se ejecuta en paralelo en múltiples UC, que pueden ser CPUs, GPUs u otros dispositivos compatibles con OpenCL.

La ejecución de los *kernels* en OpenCL se organiza en términos de una jerarquía de trabajo. La unidad básica de ejecución es el grupo de trabajo (WG, por sus siglas en inglés), que está compuesto por un conjunto de elementos de trabajo (WI, por sus siglas en inglés). Estos últimos son instancias individuales de un *kernel* que se ejecutan de manera concurrente dentro de un WG. Un WG se ejecuta en un dispositivo de cómputo específico y puede estar formado por uno o más WI.

OpenCL ofrece dos modelos de ejecución principales: el modelo de tarea (*task*) y el modelo de rango de índices numéricos (*ND-range*).

En el *ND-range*, los WI se organizan en una estructura de índices multidimensional que puede tener entre una y tres dimensiones, dependiendo de la configuración específica del *kernel* y del dispositivo de cómputo. Esta estructura permite la ejecución paralela en la que

```
// Kernel
__global__ void add(int n, float *x, float *y)
{
  int index = threadIdx.x;
  int stride = blockDim.x;

  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}

// Declaración de variables
float *x, *y;

// Asignar memoria en la GPU
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));

// Inicializar los datos en la CPU

// Lanzar el kernel en la GPU
add<<<1, 256>>>(N, x, y);

// Esperar a que la GPU termine antes de acceder a los datos en el host
cudaDeviceSynchronize();

// Liberar la memoria
cudaFree(x);
cudaFree(y);
```

Figura 2.10: Ejemplo de suma de vectores en CUDA.

cada WI realiza un cálculo independiente.

Por otro lado, en el modelo *task*, que se utiliza principalmente en FPGAs y CPUs, en lugar de dividir la ejecución en WI dentro de un espacio de índices, se divide en tareas individuales. Cada tarea representa una unidad de trabajo independiente que puede asignarse a un procesador específico.

El modelo *task* es especialmente adecuado para dispositivos como FPGAs y CPUs, donde la granularidad de la ejecución puede ser más flexible y se pueden aprovechar mejor las características específicas del hardware. En cambio, el modelo de *ND-range* se utiliza con mayor frecuencia en GPUs, donde la ejecución masivamente paralela de WI en un espacio de índices establecido es altamente eficiente.

El modelo de ejecución de OpenCL también proporciona mecanismos para la sincronización y la comunicación entre los WI. Para esto, pueden utilizar barreras de sincronización que aseguran que todos los WI dentro de un WG hayan alcanzado un punto de sincronización antes de continuar con la ejecución. Además, se pueden utilizar objetos de memoria compartida para permitir una comunicación eficiente y una cooperación entre los WI dentro de un WG.

En la Figura 2.11 se ilustra un ejemplo de suma de vectores en OpenCL. A simple vista, se puede notar que el código OpenCL requiere más líneas de código que CUDA (aproximdamente un 133 % más), para resolver el mismo problema. Este hecho es una muestra de la mayor verbosidad y complejidad estructural en el código OpenCL para realizar operaciones equivalentes a CUDA.

2.2.2.3. Modelo de memoria

OpenCL organiza los datos a través de objetos de memoria que pueden ser accedidos por los *kernels* durante la ejecución. Los objetos de memoria se dividen en diferentes tipos,

como memoria global, memoria local y memoria privada:

- Memoria global: es accesible por todos los WI y se utiliza para almacenar datos que deben ser compartidos entre diferentes WG. La memoria global es persistente y puede ser leída y escrita de manera eficiente por todos los WI.
- Memoria local: es utilizada por los WI dentro de un mismo WG y se emplea para almacenar datos que son compartidos y utilizados de manera colaborativa por los WI dentro de ese WG. En dispositivos como la GPU, la memoria local permite una comunicación eficiente y una reducción de la carga de acceso a la memoria global.
- Memoria privada: es utilizada por cada WI de manera exclusiva y se emplea para almacenar datos privados que son necesarios para los cálculos individuales de cada WI. La memoria privada es rápida pero de tamaño limitado, y se utiliza para minimizar el acceso a la memoria global.

El modelo de memoria de OpenCL también proporciona mecanismos para la transferencia de datos entre la CPU y los dispositivos. Los datos se pueden transferir de manera explícita utilizando operaciones de lectura y escritura, y OpenCL proporciona funciones para gestionar la copia eficiente de datos entre el *host* y los dispositivos.

2.2.2.4. Comparación entre CUDA y OpenCL

La comparación entre CUDA y OpenCL puede darse en diferentes aspectos:

- Portabilidad: OpenCL es una opción más flexible, debido a que es universalmente adoptable y compatible con una amplia variedad de dispositivos de diferentes fabricantes. Por otro lado, CUDA es una tecnología propietaria de NVIDIA y solo puede utilizarse con GPUs de esta compañía.
- Madurez y herramientas de desarrollo: CUDA cuenta con un conjunto de herramientas más maduras y robustas debido a su tiempo en el mercado y al hecho de ser desarrollado por un solo proveedor. Esto incluye capacidades de depuración, perfilado y documentación extensa. Por otro lado, OpenCL, al ser abierto y soportado por una comunidad más dispersa, tiene un desarrollo más fragmentado de sus herramientas no alcanzan el nivel de sofisticación de las de CUDA.
- Facilidad de aprendizaje y uso: CUDA se destaca por tener un lenguaje de programación y un modelo de programación intuitivos y fáciles de aprender, orientado al lenguaje C/C++, mientras que OpenCL tiene una curva de aprendizaje más pronunciada debido a su modelo de programación de bajo nivel y su abstracción de hardware más general, lo que requiere un entendimiento más profundo de la arquitectura subyacente de los dispositivos.

2.2.3. OpenMP

OpenMP ofrece un enfoque flexible y portátil para aprovechar el paralelismo en arquitecturas *multicore*, permitiendo a los desarrolladores aprovechar al máximo el rendimiento de los sistemas modernos [13].

```c
#include <CL/cl.h>

// Kernel
__kernel void add(int n, __global float *x, __global float *y) {
    int index = get_global_id(0);
    if (index < n)
        y[index] = x[index] + y[index];
}

// Declaración de variables
float *x = (float*)malloc(sizeof(float) * N);
float *y = (float*)malloc(sizeof(float) * N);

// Inicializar los datos en la CPU

// Configurar el entorno OpenCL (esto es un ejemplo simplificado)
cl_platform_id platform;
cl_device_id device;
cl_context context;
cl_command_queue queue;
cl_program program;
cl_kernel kernel;

clGetPlatformIDs(1, &platform, NULL);
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
queue = clCreateCommandQueue(context, device, 0, NULL);

// Crear buffers y copiar datos a la GPU
cl_mem x_buf = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float) * N, x, NULL);
cl_mem y_buf = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, sizeof(float) * N, y, NULL);

// Crear y construir el programa
program = clCreateProgramWithSource(context, 1, (const char**)&kernel_source, NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// Crear el kernel y configurar los argumentos
kernel = clCreateKernel(program, "add", NULL);
clSetKernelArg(kernel, 0, sizeof(int), &N);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &x_buf);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &y_buf);

// Lanzar el kernel
size_t global_work_size = N;
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_work_size, NULL, 0, NULL, NULL);

// Copiar los datos de vuelta a la CPU
clEnqueueReadBuffer(queue, y_buf, CL_TRUE, 0, sizeof(float) * N, y, 0, NULL, NULL);

// Liberar recursos
clReleaseMemObject(x_buf);
clReleaseMemObject(y_buf);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(queue);
clReleaseContext(context);

free(x);
free(y);
```

Figura 2.11: Ejemplo de suma de vectores en OpenCL.

2.2.3.1. Modelo de programación

El modelo de programación de OpenMP se basa en el paradigma *Single Program Multiple Data* (SPMD), lo cual implica que todos los hilos ejecutan el mismo código, pero pueden trabajar con diferentes conjuntos de datos. Las directivas de OpenMP, que definen las regiones paralelas, se insertan en el código fuente y son interpretadas por el compilador durante

el proceso de compilación. Por ejemplo, la directiva `#pragma omp parallel for` se utiliza para paralelizar un bucle *for* en lenguajes como C/C++.

Las rutinas de librería de OpenMP proporcionan funciones que permiten controlar y consultar el entorno de ejecución paralela. Por ejemplo, la función `omp_get_num_threads()` devuelve el número de hilos que están ejecutando una región paralela.

2.2.3.2. Modelo de ejecución

El enfoque de ejecución de OpenMP se basa en la idea de dividir el trabajo de un programa en tareas más pequeñas y distribuir las mismas entre múltiples hilos de ejecución. Cada hilo se encarga de ejecutar una porción del código y realizar operaciones simultáneas en diferentes conjuntos de datos. Para hacer uso de OpenMP, es necesario identificar las secciones paralelas en el código, las cuales son aquellas partes del programa donde múltiples hilos pueden ejecutarse en paralelo.

Dentro de la región paralela, se emplea una directiva de compilador para indicar las secciones del código que deben ejecutarse en paralelo. Esta directiva especifica la forma en la que se debe dividir y asignarse las tareas entre los hilos, así como también el mecanismo de sincronización cuando sea necesario. OpenMP ofrece una amplia variedad de directivas y cláusulas que permiten controlar de manera detallada y específica el comportamiento de la ejecución paralela [87].

2.2.3.3. Modelo de memoria

En cuanto al modelo de memoria, OpenMP sigue un enfoque de memoria compartida, lo cual significa que todos los hilos de ejecución comparten una región de memoria común, lo que les permite acceder y modificar datos compartidos. Sin embargo, también pueden tener su propia memoria privada para almacenar datos locales.

Para controlar el acceso a las variables compartidas y prevenir condiciones de carrera, OpenMP ofrece diversas alternativas de sincronización. Por ejemplo, las secciones críticas (definidas con la directiva *critical*) aseguran que únicamente un hilo a la vez ejecute un bloque de código, mientras que las barreras (definidas con *barrier*) sincronizan a todos los hilos en un punto específico del programa.

2.2.3.4. Soporte para programación heterogénea

OpenMP también ofrece soporte para programación heterogénea, lo que permite aprovechar el paralelismo en sistemas con otros aceleradores y produce una mayor capacidad de procesamiento y rendimiento en aplicaciones que requieren un alto nivel de cómputo.

Para habilitar la programación heterogénea en OpenMP, se utilizan directivas fundamentales que permiten especificar las tareas que se ejecutarán en la CPU y las que se ejecutarán en el acelerador. Estas directivas incluyen:

- *target*: esta directiva indica que un bloque de código debe ser ejecutado en un dispositivo objetivo, como una GPU. Se utiliza para identificar las secciones del código que se pueden acelerar en paralelo utilizando el poder de cómputo del dispositivo.
- *target data*: esta directiva se utiliza para especificar la transferencia de datos entre la CPU y el dispositivo. Permite mover los datos necesarios desde la memoria principal del sistema hacia la memoria del dispositivo objetivo antes de la ejecución y viceversa, después de que se haya completado.

- *target teams*: esta directiva se utiliza para crear equipos de hilos que se ejecutarán en el dispositivo objetivo. Cada equipo de hilos se asigna a una región paralela y se ejecuta de forma concurrente en el dispositivo objetivo.
- *target parallel*: esta directiva se utiliza para identificar una región paralela que se ejecutará en el dispositivo objetivo. Los hilos de ejecución dentro de esta región se distribuyen para aprovechar el paralelismo en el dispositivo.

Aunque OpenMP ofrece soporte para programación heterogénea, es importante tener en cuenta algunas limitaciones significativas al utilizar esta API en estos entornos. Al estar centrado en la CPU, junto con un soporte limitado para GPUs y otros aceleradores, OpenMP conduce a un rendimiento que no es el óptimo en plataformas que requieren una gestión más compleja y especializada de la memoria y la concurrencia. A pesar de que OpenMP evolucionó para incluir algunas características de computación heterogénea, carece de control a bajo nivel y de la flexibilidad que ofrecen modelos como OpenCL o CUDA, lo que restringe su efectividad en entornos con diferentes dispositivos.

En la Figura 2.12 se ilustra un ejemplo de suma de vectores en OpenMP.

```
1  #include <omp.h>
2
3  // Kernel
4  void add(int n, float *x, float *y)
5  {
6    #pragma omp target teams distribute parallel for map(to: x[0:n]) map(tofrom: y[0:n])
7    for (int i = 0; i < n; i++)
8      y[i] = x[i] + y[i];
9  }
10
11 // Declaración de variables
12 float *x = new float[N];
13 float *y = new float[N];
14
15 // Inicializar los datos en la CPU
16
17 // Lanzar el kernel
18 add(N, x, y);
19
20 // Liberar la memoria
21 delete[] x;
22 delete[] y;
23
24 return 0;
```

Figura 2.12: Ejemplo de suma de vectores en OpenMP.

2.2.4. OpenACC

OpenACC fue desarrollado en 2011 como un modelo de programación basado en directivas, al igual que OpenMP, para aprovechar el paralelismo en el código, lo que permite a los compiladores construir código optimizado para una variedad de aceleradores y lograr un alto rendimiento en estas diversas arquitecturas. OpenACC surge debido a la proliferación de GPUs y arquitecturas *many-core* en el ámbito del HPC. Busca proporcionar un modelo de programación eficiente y que al mismo tiempo permita la portabilidad a otras arquitecturas y la interoperabilidad con otras tecnologías de programación paralela como OpenMP y CUDA. Esto facilita la integración de OpenACC en aplicaciones que ya están basadas en estas tecnologías, ampliando su aplicabilidad y eficiencia [14].

2.2.4.1. Modelo de plataforma

OpenACC está diseñado para ser compatible con una amplia gama de plataformas, incluyendo sistemas con GPUs de NVIDIA y de AMD. La portabilidad de OpenACC se logra mediante la implementación de compiladores y librerías específicas para cada plataforma. Estas implementaciones se encargan de traducir las directivas de OpenACC en código ejecutable adecuado para la plataforma objetivo. Además, OpenACC proporciona cláusulas de ajuste de rendimiento que permiten a los desarrolladores controlar el mapeo de los datos y las tareas en el dispositivo, optimizando así el rendimiento para la arquitectura específica.

Es importante destacar que OpenACC también puede combinarse con otros enfoques de programación paralela, como OpenMP y CUDA, para aprovechar tanto el paralelismo de memoria compartida como el de memoria distribuida, y proporcionar un enfoque flexible y escalable para la programación de sistemas acelerados por GPU.

2.2.4.2. Modelo de ejecución

El modelo de ejecución en OpenACC se fundamenta en la idea de identificar secciones de código que pueden ser ejecutadas en paralelo. Estas secciones, conocidas como regiones paralelas, son definidas mediante directivas específicas del compilador OpenACC. Dentro de estas regiones, el código es ejecutado en paralelo en distintas unidades de procesamiento, tales como núcleos de CPU o GPU.

Una característica clave del modelo de ejecución de OpenACC es su capacidad para que los programadores especifiquen los bucles o secciones de código que deben ser ejecutados en paralelo y aquellos que deben ser ejecutados de manera secuencial. Esto se logra mediante el uso de directivas de compilador como *parallel* y *kernels*.

Además, el modelo de ejecución de OpenACC proporciona mecanismos para controlar la asignación de tareas a los diferentes recursos de procesamiento. Esto se logra mediante el uso de directivas como *loop*, que permite al programador especificar el mecanismo de distribución de los bucles entre los hilos de ejecución, o *barrier*, que permiten sincronizar la ejecución paralela en diferentes puntos del programa [88].

En la Figura 2.13 se ilustra un ejemplo de suma de vectores en OpenACC.

2.2.4.3. Modelo de memoria

En cuanto al modelo de memoria, OpenACC sigue un enfoque de memoria compartida virtual, lo cual significa que los datos se comparten entre la CPU y el acelerador mediante transferencias automáticas y transparentes realizadas por el compilador. OpenACC utiliza directivas de datos para indicar qué datos deben estar presentes en el acelerador y cuándo deben transferirse entre la CPU y el dispositivo.

El modelo de memoria de OpenACC también permite la gestión de la memoria global del acelerador. Los datos se pueden alojar en la memoria global del mismo y se pueden acceder y modificar tanto por la CPU como por el acelerador. Esto facilita el intercambio de datos entre ambos y permite una cooperación eficiente entre las dos unidades de procesamiento.

2.2.5. Kokkos

Kokkos [89], introducido en 2014, es una librería que implementa un modelo de programación y ofrece abstracciones para la programación paralela. Diseñado para ser ejecutado en múltiples tipos de hardware, incluyendo CPUs *multicore*, GPUs y otros aceleradores,

```
#include <openacc.h>

// Kernel
void add(int n, float *x, float *y)
{
  #pragma acc kernels loop copyin(x[0:n]) copyout(y[0:n])
  for (int i = 0; i < n; i++)
    y[i] = x[i] + y[i];
}

// Declaración de variables
float *x = new float[N];
float *y = new float[N];

// Inicializar los datos en la CPU

// Lanzar el kernel en la GPU
add(N, x, y);

// Liberar la memoria
delete[] x;
delete[] y;

return 0;
```

Figura 2.13: Ejemplo de suma de vectores en OpenACC.

Kokkos facilita la escritura de código paralelo y portable, manteniendo al mismo tiempo un alto rendimiento en diversas plataformas.

2.2.5.1. Modelo de plataforma

En Kokkos, el código se escribe en C++ estándar, pero introduce abstracciones y extensiones propias para la paralelización y gestión de la memoria. La librería se encarga de la portabilidad y el mapeo eficiente en diferentes plataformas. Esto incluye el uso de características avanzadas de C++ para abstraer y gestionar la paralelización y la asignación de memoria.

Kokkos proporciona un conjunto de herramientas y librerías que ayudan en el desarrollo de aplicaciones paralelas, permitiendo que el mismo código base se ejecute eficientemente en diferentes tipos de hardware.

2.2.5.2. Modelo de ejecución

El modelo de ejecución de Kokkos se centra en *Functors* o *Lambdas*, que son objetos o funciones que encapsulan el código a ejecutar en paralelo. En lugar de utilizar hilos y bloques como en CUDA, Kokkos utiliza una abstracción llamada *Execution Spaces* para definir el dispositivo donde se ejecutará el código paralelo.

Los patrones de ejecución comunes en Kokkos incluyen paralelización por bucles `for` (a través de la instrucción `Kokkos::parallel_for`) y reducciones (a través de la instrucción `Kokkos::parallel_reduce`). Estas construcciones permiten a los desarrolladores especificar la concurrencia y la organización de datos sin depender de la arquitectura específica de hardware.

En la Figura 2.14 se ilustra un ejemplo de suma de vectores en Kokkos.

2.2.5.3. Modelo de memoria

Kokkos introduce un modelo de memoria avanzado que permite la portabilidad entre diferentes arquitecturas de memoria. Utiliza abstracciones como *View* para administrar el acceso a la memoria y las transferencias de datos entre diferentes espacios de memoria. Las *Views* son similares a los punteros multidimensionales y proporcionan una interfaz consistente para acceder a los datos, independientemente del dispositivo subyacente.

El modelo de memoria de Kokkos administra de forma transparente las diferencias entre las memorias de los dispositivos y la memoria del *host*, facilitando así la escritura de código portable.

```
1  #include <Kokkos_Core.hpp>
2
3  // Kernel
4  void add(int n, float *x, float *y)
5  {
6    Kokkos::parallel_for("add_vectors", n, KOKKOS_LAMBDA(const int i) {
7      y[i] = x[i] + y[i];
8    });
9    Kokkos::fence();
10 }
11
12 Kokkos::initialize(argc, argv);
13
14 // Declaración de variables
15 float *x = new float[N];
16 float *y = new float[N];
17
18 // Inicializar los datos en la CPU
19
20 // Lanzar el kernel en la GPU
21 add(N, x, y);
22
23 // Liberar la memoria
24 delete[] x;
25 delete[] y;
26
27 Kokkos::finalize();
```

Figura 2.14: Ejemplo de suma de vectores en Kokkos.

2.2.6. RAJA

RAJA, similar a Kokkos, fue presentado por primera vez en 2014 y proporciona una librería de abstracción sobre diferentes modelos de programación paralela, enfocándose específicamente en la optimización de bucles para maximizar el uso de los recursos de hardware disponibles. Está diseñado para su uso en sistemas que incluyen tanto CPUs como GPUs, ofreciendo un enfoque coherente y unificado para la programación paralela [90].

2.2.6.1. Modelo de plataforma

RAJA está diseñado para ser independiente de la plataforma, permitiendo a los desarrolladores escribir código paralelo ejecutable en distintas arquitecturas de hardware. Esto se logra mediante la abstracción de los detalles específicos de la plataforma y proporcionando una interfaz coherente para la ejecución de bucles paralelos. RAJA soporta una variedad de *backends*, como ejecución secuencial, optimizaciones SIMD y *multithreading* de CPU con OpenMP, y CUDA, lo que refleja su diseño para adaptarse a diferentes entornos de cómputo.

2.2.6.2. Modelo de ejecución

El modelo de ejecución de RAJA se centra en la abstracción de patrones de bucles, utilizando un conjunto de directivas de preprocesador y plantillas de C++ para convertir bucles secuenciales en paralelos. Define *execution policies* que determinan la forma en la que se deben ejecutar los bucles, adaptándose a distintas arquitecturas de hardware y permitiendo que el mismo código se ejecute tanto en CPUs como en el acelerador. Estas políticas abarcan diferentes tipos de ejecución, como secuencial, OpenMP, CUDA, entre otros, y pueden ser combinaciones complejas de políticas más simples, lo que brinda una flexibilidad significativa para acceder a características avanzadas de los modelos de programación paralela [91].

Los patrones de ejecución en RAJA abarcan bucles `for` paralelos, reducciones y operaciones de barrido. Los desarrolladores pueden seleccionar entre diversas políticas de ejecución, adaptándolas a la arquitectura de hardware objetivo, como la paralelización con OpenMP para CPUs y CUDA para GPUs.

```
1   #include <RAJA/RAJA.hpp>
2
3   // Kernel
4   void add(int n, float *x, float *y)
5   {
6     RAJA::forall<RAJA::cuda_exec<256>>(RAJA::RangeSegment(0, n), [=] RAJA_DEVICE (int i) {
7       y[i] = x[i] + y[i];
8     });
9   }
10
11  // Declaración de variables
12  float *x = new float[N];
13  float *y = new float[N];
14
15  // Inicializar los datos en la CPU
16
17  // Lanzar el kernel en la GPU
18  add(N, x, y);
19
20  // Liberar la memoria
21  delete[] x;
22  delete[] y;
```

Figura 2.15: Ejemplo de suma de vectores en RAJA.

2.2.6.3. Modelo de memoria

RAJA también implementa un modelo de memoria avanzado para facilitar la gestión de datos en arquitecturas de memoria heterogéneas. Aunque se centra más en la abstracción de la ejecución de bucles, RAJA proporciona mecanismos para administrar la asignación y el movimiento de datos entre distintos espacios de memoria, como la memoria del *host* y la del dispositivo. Este modelo incluye el concepto de espacios de iteración, que definen conjuntos de índices de bucle para un *kernel*, asegurando operaciones de acceso indexadas constantes en tiempo y portables entre diferentes tipos de ejecución.

En la Figura 2.15 se ilustra un ejemplo de suma de vectores en RAJA.

2.3. SYCL

SYCL es un modelo de programación abstracto y multiplataforma en C++ diseñado específicamente para la computación heterogénea. Su desarrollo surge debido a la necesidad de ofrecer una alternativa que combine la portabilidad y eficiencia de APIs paralelas como OpenCL con la facilidad de uso y flexibilidad del C++ moderno, lo que permite a los desarrolladores escribir código estándar en este lenguaje utilizando técnicas habituales de alto nivel, al tiempo que acceden a un amplio rango de capacidades de implementaciones subyacentes como OpenCL [17].

A diferencia de otras soluciones como CUDA, que requieren un lenguaje de programación específico o un entorno de ejecución particular, SYCL se integra de manera más orgánica en el flujo de trabajo de desarrollo en C++, lo que facilita su adopción en proyectos existentes. Por otro lado, en contraste con soluciones para la computación heterogénea como OpenACC, que se centra en la simplicidad mediante directivas de compilación, SYCL ofrece una mayor flexibilidad y control detallado de los dispositivos, permitiendo a los desarrolladores aprovechar mejor las capacidades del hardware y personalizar el cómputo paralelo según las necesidades específicas de sus aplicaciones.

2.3.1. Modelo de plataforma

SYCL proporciona un modelo de plataforma que permite a los desarrolladores aprovechar eficientemente los recursos de cómputo disponibles en diferentes dispositivos. Este modelo se basa en una jerarquía de dispositivos que permite escribir código portable que se ejecuta de manera eficiente en diferentes tipos de hardware. La Figura 2.16 ilustra el modelo de plataforma de SYCL.

El modelo de plataforma en SYCL se compone de tres elementos principales:

- *Hosts*: el *host* es el procesador principal de la máquina y es responsable de coordinar y gestionar las operaciones de cómputo en los dispositivos.
- Dispositivos: son los recursos de cómputo específicos, como las GPU o las FPGAs, que se utilizan para ejecutar el código paralelo.
- Colas de comandos: actúan como intermediarios entre el *host* y los dispositivos, y permiten la transferencia de datos y la ejecución de comandos en los dispositivos.

EL modelo de plataforma de SYCL se fundamenta en el modelo de OpenCL, que se compone de un *host* que se conecta a uno o varios dispositivos heterogéneos. Para poder operar en un dispositivo o grupo de dispositivos, se construye un contexto SYCL que contiene toda la información de tiempo de ejecución necesaria para el *runtime* de SYCL y la API *backend*.

2.3.1.1. Compilación y ejecución

El proceso de compilación de SYCL se ilustra en la Figura 2.17. Comienza con las Unidades de Traducción C++ de SYCL, que son procesadas por un controlador de compilador, generalmente basado en *Clang*. El controlador identifica los *kernels* destinados a la ejecución en los dispositivos y los separa del código del *host*. Antes de la compilación final, los *kernels* son convertidos a un formato de representación intermedia (IR, por sus siglas en inglés), el cual resulta fundamental para su posterior ejecución en el dispositivo. El IR posibilita la optimización y adaptación del código a diversas arquitecturas de dispositivos. Estos kernels, ahora en formato IR, necesitan ser compilados para la arquitectura del dispositivo objetivo, lo que puede realizarse dentro del mismo proceso de compilación o como parte de una

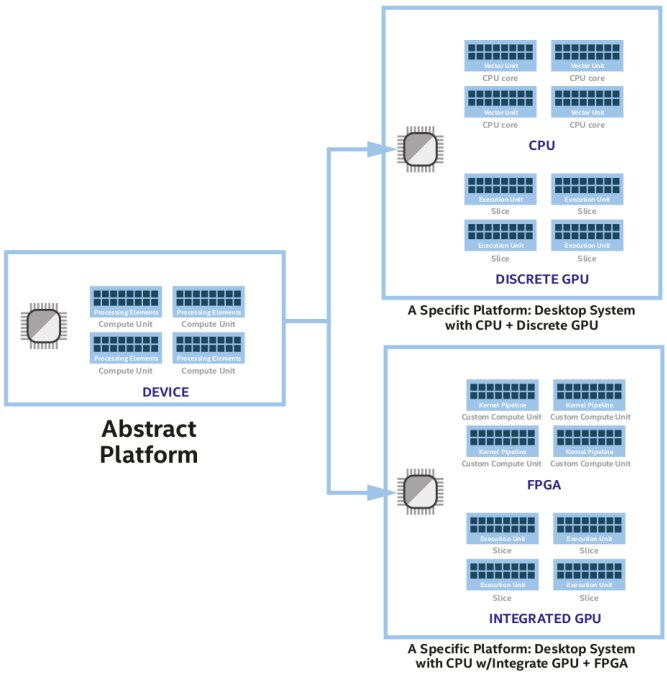

Figura 2.16: Modelo de plataforma de SYCL. Puede ser usado de forma abstracta o de forma explícita (extraído de [92]).

invocación de proceso separada. Paralelamente, el código del *host* junto con algún código y/o librerías del *runtime* de SYCL se compilan para la arquitectura de la CPU del *host*. Finalmente, todos los *kernels* para todos los *backends* se combinan con el código del *host* y el *runtime* para generar un programa binario ejecutable de SYCL, dirigido a los dispositivos. Si se quiere apuntar a varios *backends*, este proceso puede necesitar repetirse para cada uno.

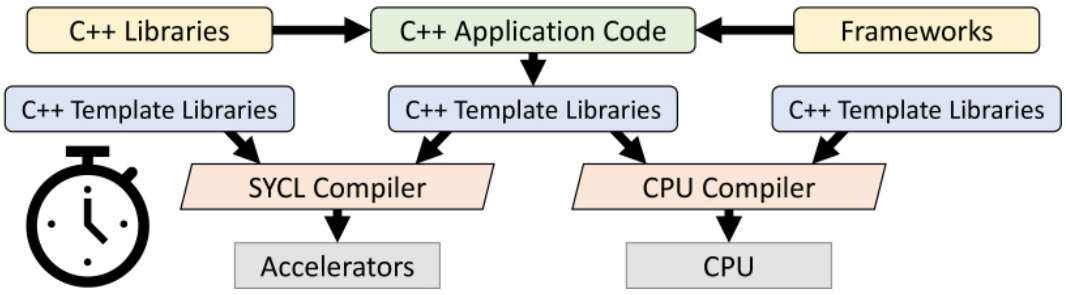

Figura 2.17: Proceso de compilación en SYCL (extraído de [93]).

2.3.1.2. Backend SYCL

SYCL tiene un enfoque de programación genérico para el lenguaje C++ que permite apuntar a diversas APIs heterogéneas. Las implementaciones de SYCL habilitan estas APIs objetivo mediante la implementación de *backends* específicos de SYCL, donde cada uno

ofrece su propio documento que define las APIs correspondientes y cómo acceder a las versiones del dispositivo y la plataforma.

2.3.2. Modelo de ejecución

El modelo de ejecución de SYCL puede dividirse en dos: el modelo de ejecución de la aplicación y el modelo de ejecución del *kernel*.

2.3.2.1. Modelo de Ejecución de la Aplicación SYCL

En la aplicación SYCL, la gestión y organización de la ejecución de *kernels* se logra mediante su agrupación con los requisitos asociados en un objeto de grupo de comandos (GC) [8]. Estos objetos de GC son procesados a través de una cola que especifica el dispositivo donde se ejecutará el *kernel*, teniendo la posibilidad de asignar un único objeto de GC a múltiples colas. Cuando se introduce un GC en una cola de SYCL, se identifican y capturan todos los requisitos necesarios para la ejecución del *kernel*. Este enfoque permite que la ejecución de un *kernel* comience inmediatamente después de que se hayan cumplido sus requisitos previos. El modelo de ejecución adoptado por SYCL se enfoca en la coordinación y ejecución eficiente de los *kernels*, lo que contribuye significativamente a mejorar el rendimiento y la eficiencia en las aplicaciones.

Los recursos gestionados por SYCL son:

- Plataformas: una plataforma representa una implementación específica que proporciona acceso a los recursos de cómputo. Puede haber múltiples plataformas disponibles en un sistema, cada una de las cuales puede tener diferentes dispositivos asociados. Las plataformas se utilizan para enumerar y seleccionar los dispositivos disponibles en el sistema.

- Contextos: un contexto es una abstracción que encapsula los recursos relacionados con la ejecución de tareas. Cada contexto está asociado con una o más plataformas y es utilizado para crear y gestionar los dispositivos, colas de comandos y *kernels*. Los contextos permiten la coordinación entre los diferentes elementos del modelo de plataforma.

- Dispositivos: los dispositivos representan los recursos de cómputo específicos, como las GPU, las FPGAs o las CPUs, que se utilizan para ejecutar el código paralelo. Cada dispositivo está asociado con un contexto y puede tener múltiples colas de comandos asociadas. Los dispositivos pueden ser seleccionados y configurados para aprovechar las capacidades específicas de hardware disponibles.

- *Kernels*: un *kernel* en SYCL es una función que define las operaciones de cómputo que se realizarán en los datos. Los *kernels* son escritos en un lenguaje de programación compatible con SYCL, como C++ o un subconjunto de OpenCL C, y se ejecutan en los dispositivos. Los *kernels* se pueden compilar para diferentes dispositivos y se pueden reutilizar en distintos contextos.

- Colas de comandos: actúan como intermediarios entre el *host* y los dispositivos. Permiten la transferencia de datos entre ambos, así como la ejecución de *kernels* en los

[8] Un grupo de comandos es una abstracción que encapsula un conjunto de operaciones y requisitos, como la ejecución de un *kernel* y operaciones de memoria, para su procesamiento coordinado en un dispositivo de cómputo.

dispositivos. Las colas de comandos son responsables de la planificación y la coordinación de las operaciones de cómputo en los dispositivos, y pueden ser utilizadas para lograr una ejecución paralela eficiente.

Gestión de recursos de *backend* por la aplicación SYCL: el *runtime* de SYCL se encarga de administrar los recursos necesarios para la API de *backend*, los cuales son utilizados para manejar los dispositivos heterogéneos a los que se tiene acceso. Estos recursos incluyen controladores, *pools* de memoria, colas de comando y otros objetos temporales. La interfaz de programación de SYCL se encarga de gestionar la duración de estos recursos, siguiendo las reglas de *Resource Acquisition Is Initialization* [9].

Grupos de comandos y orden de ejecución en SYCL: de manera predeterminada, las colas de SYCL ejecutan las funciones de *kernel* de forma desordenada, teniendo en cuenta la información de dependencia. Los desarrolladores solo deben especificar los datos necesarios para ejecutar un *kernel* en particular. El *runtime* de SYCL se encarga de garantizar que los *kernels* se ejecuten en un orden que asegure su corrección. Luego de especificar los modos de acceso y los tipos de memoria, se construye un grafo de dependencias acíclico dirigido de *kernels* en tiempo de ejecución. La Figura 2.18 muestra un ejemplo básico de un grafo de dependencias.

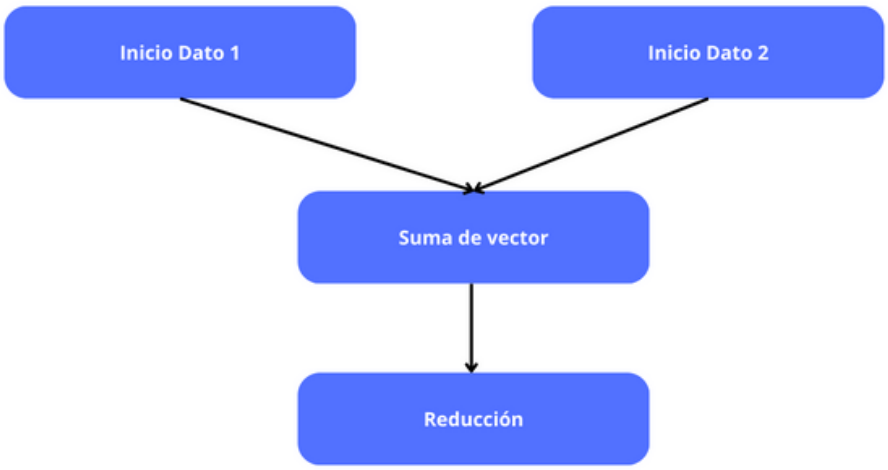

Figura 2.18: Grafo de dependencias del patrón "Y".

2.3.2.2. Modelo de Ejecución del *Kernel* en SYCL

Cuando se envía un *kernel* para su ejecución, se establece un espacio de índices donde cada punto en ese espacio corresponde a una instancia del cuerpo del *kernel* que se ejecuta. Estas instancias del *kernel* son los WI, que se identifican por su posición en el espacio de índices, lo cual les proporciona un identificador global. Aunque todos los WI ejecutan el

[9] *Resource Acquisition Is Initialization* (RAII) es un concepto de programación en el que la adquisición de recursos (como memoria, recursos de red o archivos) se realiza durante la inicialización de un objeto, garantizando que todos los recursos sean liberados cuando la vida del objeto termina, para evitar fugas de recursos y otros errores relacionados.

mismo código, el camino de ejecución y los datos procesados pueden variar utilizando el identificador global para personalizar el cálculo. Todos los aspectos de la ejecución del *kernel* siguen su curso normal, con la excepción de que la función del *kernel* en sí no se ejecuta. Existen diferentes tipos de kerneles:

- *Kernels* Básicos: SYCL proporciona un enfoque directo para la ejecución de código paralelo mediante la utilización de *kernels*. Estos *kernels* operan sobre un espacio de índices tridimensional, definido por el objeto $range < N >$, donde N representa el número de dimensiones que puede ser 1, 2, o 3. Dentro de un *kernel*, cada WI se ejecuta de forma independiente y es identificado por un valor del tipo $item < N >$. Este tipo $item < N >$ incluye dos componentes clave: un identificador de trabajo, del tipo $id < N >$, que define la posición única del WI en el espacio de índices, y un $range < N >$ que especifica la cantidad total de WIs participando en la ejecución del *kernel*. De este modo, cada WI en un *kernel* de SYCL tiene un contexto claro tanto de su posición individual como de su relación con el conjunto general de WIs. La Figura 2.19 muestra gráficamente este tipo de kernel.
- *Kernels ND-range*: los WIs pueden agruparse en WGs, permitiendo una organización más eficiente en la ejecución paralela y una mejor descomposición del espacio de índices. Cada WI dentro de un *ND-range* recibe un identificador único global, el cual se alinea con la dimensionalidad del espacio de índices del *ND-range*. Dentro de cada WG, los WIs también tienen un identificador local único, que es específico para ese grupo. Por lo tanto, cada WI puede ser identificado de manera única ya sea por su identificador global o por una combinación de su identificador local y el identificador del WG al que pertenece. Los WIs en un WG determinado se ejecutan en los elementos de procesamiento de una única unidad de cálculo, permitiendo la paralelización eficiente del trabajo. La Figura 2.20 ilustra la organización *ND-range*.
- *Kernels* específicos del *backend*: SYCL permite que un *backend* muestre funcionalidades predefinidas como *kernels* integrados no programables. La disponibilidad y el comportamiento de estos *kernels* integrados son determinados por el *backend* y no es necesario que sigan los modelos de ejecución y memoria de SYCL. Además, la interfaz expuesta que utiliza estos *kernels* integrados también es específica del *backend*.

2.3.3. Modelo de memoria

Dado que SYCL es un modelo de programación de fuente única, el modelo de memoria afecta tanto a la aplicación como a las partes del *kernel*. En la aplicación, el *runtime* de SYCL se asegura de que los datos estén disponibles para la ejecución de los *kernels*, mientras que en el *kernel*, las reglas *backend* que describen el comportamiento de la memoria en un dispositivo específico se mapean a constructores de SYCL C++, haciendo posible programar *kernels* de manera eficiente en C++ puro.

El modelo de memoria en SYCL se estructura en varias capas:

- Memoria global: es accesible por todos los dispositivos y el *host*.
- Memoria constante: optimizada para lecturas frecuentes y escrituras ocasionales, es ideal para almacenar datos que no cambian durante la ejecución de un kernel.

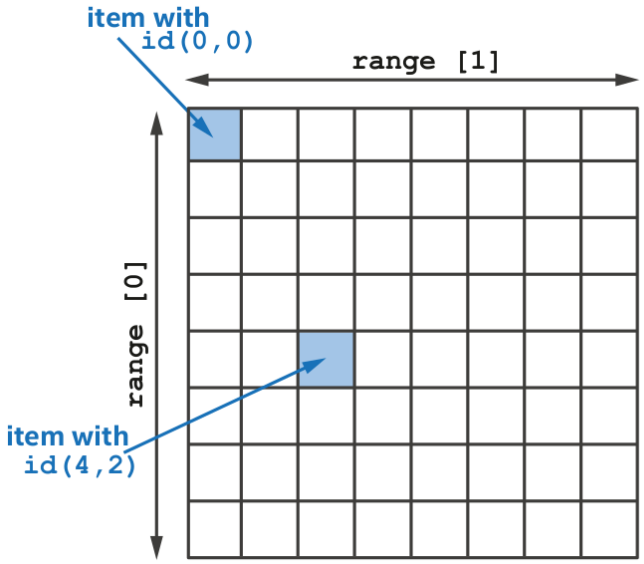

Figura 2.19: Espacio de ejecución de un *kernel* paralelo básico, mostrado para un rango 2D de 64 elementos (extraído de [92]).

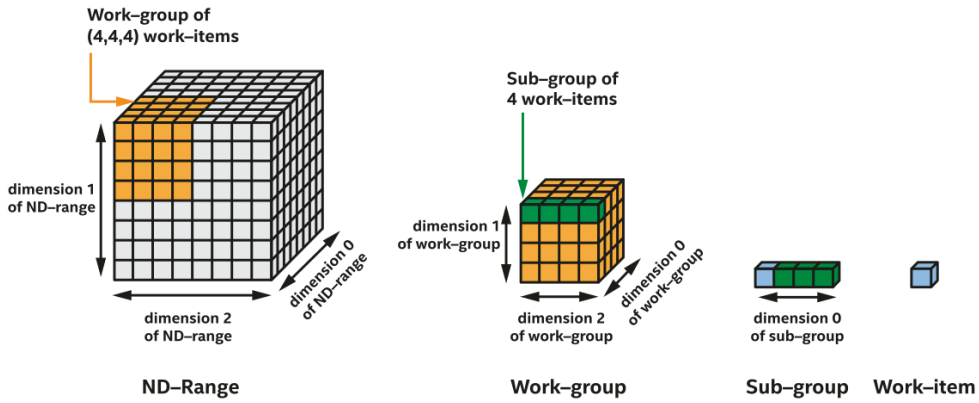

Figura 2.20: *ND-range* de 3 dimensiones divididas en WG, subgrupos y WI (extraído de [92]).

- Memoria local: es específica de un WG y, como en el caso de las GPUs, ofrece un acceso más eficiente en comparación con la memoria global. Es compartida por todos los WI dentro de WG.
- Memoria privada: es exclusiva de un único WI y se utiliza para datos que no necesitan ser compartidos con otros WIs.

SYCL también introduce dos conceptos adicionales: los *buffers* y *Unified Shared Memory* (USM, por sus siglas en inglés).

2.3.3.1. *Buffers* y *Accessors*

Un *buffer* en SYCL es un objeto que representa una región de memoria en el sistema y puede ser utilizado para almacenar datos que serán procesados por los dispositivos. Los *buffers* proporcionan una abstracción de alto nivel que permite a los desarrolladores acceder y manipular los datos de manera eficiente en diferentes dispositivos.

Para acceder a los datos en un *buffer*, se utilizan *accessors*, que es un objeto que permite tanto la lectura como la escritura de datos en un buffer. Los *accessors* proporcionan una interfaz para acceder a los datos de manera segura y eficiente desde los dispositivos de cómputo.

SYCL ofrece distintos tipos de *accessors* para adaptarse a diferentes necesidades de acceso a los datos:

- `sycl::accessor<DataType, Dimensions, AccessMode, AccessTarget>`: este es el *accessor* más básico que proporciona acceso multidimensional a los datos en un *buffer*. Los parámetros *DataType* y *Dimensions* especifican el tipo de dato y el número de dimensiones de los datos, respectivamente. *AccessMode* define si el *accessor* se utilizará para lectura, escritura o ambas operaciones. *AccessTarget* indica si el *accessor* se utilizará en el *host* o en los dispositivos.

- `sycl::accessor<DataType, Dimensions, AccessMode, AccessTarget, access::placeholder::true_t>`: este tipo de *accessor* se utiliza para especificar un *placeholder* en lugar de un *buffer* real. Los accessors de placeholder se utilizan cuando se desea definir un acceso a los datos antes de que se haya creado el buffer subyacente.

- Otros tipos de accessors: SYCL también proporciona *accessors* especializados para casos específicos, como `sycl::accessor<DataType, 1, AccessMode, AccessTarget, access::target::local>` para acceder a la memoria local dentro de un WG.

Los *accessors* en SYCL permiten un acceso controlado y eficiente a los datos en los *buffers*, asegurando la consistencia y la sincronización adecuada entre los dispositivos y el *host*.

En la Figura 2.21 se ilustra un ejemplo de suma de vectores en SYCL usando *buffers*.

2.3.3.2. USM

USM es un enfoque alternativo de gestión de memoria en SYCL. Con el USM, los desarrolladores pueden utilizar un modelo de memoria unificado en el que los datos son accesibles y compartidos de manera transparente entre el *host* y los dispositivos de cómputo.

En este mecanismo, los datos se alojan en un espacio de memoria unificado que es accesible tanto por el *host* como por los dispositivos. Esto elimina la necesidad de transferir explícitamente los datos entre ambos mediante *buffers*, ya que estos se encuentran automáticamente disponibles en todos los contextos de ejecución.

SYCL proporciona 3 diferentes mecanismos para trabajar con USM:

- *USM host*: la memoria puede ser accedida tanto por el *host* como por el dispositivo, y es gestionada por el *runtime* de SYCL.

- *USM device*: la memoria es accesible solo por el dispositivo, generalmente ofreciendo mayor rendimiento pero sin acceso directo desde el *host*.

- *USM shared*: es un mecanismo intermedio, donde la memoria es accesible tanto por el *host* como por el dispositivo y las transferencias son transparentes al programador.

En definitiva, USM proporciona una forma más flexible y conveniente de trabajar con la memoria compartida entre el *host* y los dispositivos, simplificando el código y mejorando la portabilidad de las aplicaciones.

En la Figura 2.22 se ilustra un ejemplo de suma de vectores en SYCL usando USM.

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

// Kernel
void add(cl::sycl::handler& cgh, int n, cl::sycl::buffer<float, 1>& x_buf, cl::sycl::buffer<float, 1>& y_buf) {
  auto x_acc = x_buf.get_access<cl::sycl::access::mode::read>(cgh);
  auto y_acc = y_buf.get_access<cl::sycl::access::mode::read_write>(cgh);

  cgh.parallel_for<class add_vectors>(cl::sycl::range<1>(n), [=](cl::sycl::id<1> i) {
    y_acc[i] = x_acc[i] + y_acc[i];
  });
}

// Declaración de variables
float *x = new float[N];
float *y = new float[N];

// Inicializar los datos en la CPU

// Crear un queue para un dispositivo GPU
cl::sycl::queue q(cl::sycl::default_selector{});

// Crear buffers para los datos
cl::sycl::buffer<float, 1> x_buf(x, cl::sycl::range<1>(N));
cl::sycl::buffer<float, 1> y_buf(y, cl::sycl::range<1>(N));

// Lanzar el kernel en la GPU
q.submit([&](cl::sycl::handler& cgh) {
    add(cgh, N, x_buf, y_buf);
});

// Esperar a que la GPU termine antes de acceder a los datos en el host
q.wait();

// Liberar la memoria
delete[] x;
delete[] y;
```

Figura 2.21: Ejemplo de suma de vectores en SYCL usando *buffers*.

2.3.4. Implementaciones SYCL

Existen diversas implementaciones, cada una con el objetivo de optimizar y ampliar la aplicabilidad de SYCL en diferentes plataformas y entornos. Estas implementaciones reflejan la diversidad y adaptabilidad de SYCL, resaltando su capacidad para facilitar la programación en diferentes plataformas de hardware. Entre las implementaciones más destacadas se encuentran:

- DPC++ [94]: Intel provee su implementación SYCL, conocida como DPC++ dentro de su ecosistema de programación unificado oneAPI. Permite apuntar especialmente a CPUs, GPUs y FPGAs de Intel, al mismo tiempo que ofrece una gran variedad de herramientas de desarrollo, *profiling* y optimizaciones que facilitan el desarrollo y mejoran el rendimiento de los algoritmos.
- ComputeCpp [19]: actualmente adquirida por oneAPI [10], ComputeCPP es una implementación SYCL, propiedad de Codeplay, que se destaca por su compatibilidad con

[10] `https://codeplay.com/portal/news/2023/07/07/the-future-of-computecpp`

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

// Kernel
void add(int n, float *x, float *y) {
  for (int i = 0; i < n; i++)
    y[i] = x[i] + y[i];
}

auto q = cl::sycl::queue(cl::sycl::default_selector{});

// Asignar memoria en la GPU
float *x = static_cast<float*>(malloc_shared(N*sizeof(float), q));
float *y = static_cast<float*>(malloc_shared(N*sizeof(float), q));

// Inicializar los datos en la CPU

// Lanzar el kernel en la GPU
q.submit([&](cl::sycl::handler& cgh) {
cgh.parallel_for<class add_vectors>(cl::sycl::range<1>(N), [=](cl::sycl::id<1> i) {
  add(N, x, y);
});
});

// Esperar a que la GPU termine antes de acceder a los datos en el host
q.wait();

// Liberar la memoria
free(x, q);
free(y, q);
```

Figura 2.22: Ejemplo de suma de vectores en SYCL usando USM.

una variedad de plataformas y sistemas, ofreciendo una solución comercial para la programación con SYCL.

- AdaptiveCpp [95]: previamente denominado hipSYCL/OpenSYCL [96], es una implementación que facilita la programación heterogénea basada en C++ para CPUs y GPUs. Integra el paralelismo SYCL, lo que permite la transferencia de algoritmos de C++ a una amplia gama de proveedores de CPU y GPU (como Intel, NVIDIA y AMD). En particular, un solo binario puede apuntar a diferentes plataformas de hardware o incluso a plataformas de diferentes proveedores, mediante el uso de una nueva característica propia de AdaptiveCpp (denominada *generic single-pass* [97]) que aumenta la portabilidad y la productividad al ocultar la dependencia del hardware objetivo. Específicamente, AdaptiveCpp utiliza un flujo genérico de "Un Solo Origen y Un Solo Paso del Compilador" (SSCP, por sus siglas en inglés), compilando los *kernels* en una representación IR. En tiempo de ejecución, esta representación se transforma en formatos específicos del *backend*, como PTX o SPIR-V según sea necesario. Este enfoque implica una sola invocación del compilador, analizando el código una vez, independientemente del número de dispositivos o *backends* utilizados. Además, AdaptiveCpp permite también al desarrollador indicar el flujo de compilación específico de la cadena de herramientas/*backend* (si se prefiere).

- TriSYCL [21]: propiedad de Xilinx, es una implementación de código abierto que se enfoca en proporcionar una plataforma para experimentar con extensiones y optimizaciones en SYCL, dirigido especialmente a FPGAs. Su enfoque está más orientado a la innovación y a la exploración de nuevas posibilidades dentro del marco de SYCL.

- neoSYCL [98] y Sylkan [99]: son otras implementaciones que buscan expandir la cobertura de SYCL a otros dominios, como *Vulkan+SPIR-V* para neoSYCL y *VEO* para

Sylkan.

La Figura 2.23 muestra un esquema de las 3 principales implementaciones SYCL a diciembre de 2023.

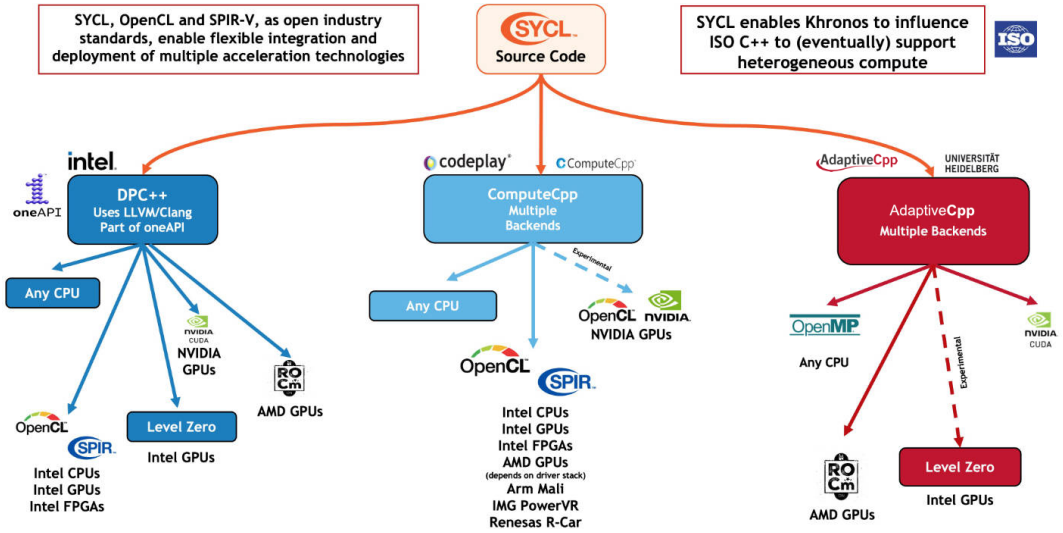

Figura 2.23: Las 3 implementaciones SYCL más destacadas: DPC++, ComputeCPP y AdaptiveCPP (extraído de [100]).

2.3.5. Comparación de los modelos de programación heterogénea

SYCL se diferencia notablemente en el ámbito de la computación heterogénea en comparación con otras tecnologías como CUDA, OpenCL, OpenMP, OpenACC, Kokkos y RAJA. A diferencia de CUDA, que está vinculado a hardware específico de NVIDIA, SYCL ofrece una portabilidad superior al ser compatible con una amplia gama de dispositivos, incluyendo GPUs, CPUs y FPGAs. Esto contrasta con la limitación inherente de hardware de CUDA, lo que brinda a SYCL una ventaja en términos de versatilidad y accesibilidad.

En comparación con OpenCL, que también es portable, pero a menudo es difícil de programar por su complejidad y bajo nivel de abstracción, SYCL proporciona una interfaz más amigable y moderna, permitiendo a los desarrolladores escribir código más legible y mantenible. Mientras que OpenCL requiere un conocimiento detallado del hardware subyacente, SYCL abstrae estos detalles, permitiendo un enfoque más centrado en el algoritmo.

Frente a OpenMP y OpenACC, que son ampliamente utilizados para el paralelismo en CPU y algunas formas de computación en GPU, SYCL ofrece una integración más cohesiva y uniforme con C++, aprovechando las características modernas del lenguaje para facilitar la escritura de programas paralelos. Aunque OpenMP y OpenACC son efectivos en sus respectivos contextos, carecen de la flexibilidad de SYCL para trabajar de manera eficiente en una variedad más amplia de plataformas y arquitecturas.

Por otro lado, Kokkos y RAJA son librerías de paralelismo en C++ diseñados para escribir código portable y eficiente para diferentes plataformas. Aunque comparten algunas similitudes con SYCL en términos de objetivos, SYCL se destaca por su capacidad de integración con el ecosistema de C++ y su cumplimiento con los estándares de la industria.

SYCL, al ser un estándar abierto desarrollado por el Khronos Group, asegura una mejor compatibilidad y soporte a largo plazo en comparación con estas soluciones más específicas.

2.4. Métricas de evaluación

En el campo de HPC, la evaluación del rendimiento de los sistemas es importante para comprender y mejorar la eficiencia de las aplicaciones. Las métricas desempeñan un papel fundamental en este proceso, debido a que proporcionan una medida objetiva y cuantitativa de la eficacia de un sistema HPC en términos de velocidad, capacidad de cálculo y portabilidad. Sin embargo, además del rendimiento, existen otros aspectos que tienen una influencia importante al momento de desarrollar una aplicación para un sistema HPC. Entre ellos, se puede mencionar: el costo de adquisición y mantenimiento de una plataforma, que son factores clave para la sustentabilidad a largo plazo de un sistema HPC; el costo y esfuerzo de programación asociado a un lenguaje, librería o metodología de desarrollo, que afectan la eficiencia en el desarrollo e implementación de aplicaciones; o la portabilidad, que determina la facilidad con la que un código puede adaptarse a diferentes entornos de hardware y software. Estos elementos, aunque no están directamente relacionados con el rendimiento bruto del sistema, son fundamentales para una evaluación integral y para tomar decisiones informadas en la selección de soluciones HPC.

2.4.1. Rendimiento

En HPC, se utiliza frecuentemente la medida de GFLOPS para evaluar el rendimiento de un sistema. Un "*flop*" representa una operación de punto flotante, siendo un GFLOP el equivalente a mil millones de estas operaciones. GFLOPS es especialmente útil para evaluar el rendimiento de sistemas que llevan a cabo operaciones intensivas en punto flotante y se calcula mediante la formula:

$$GFLOPS = \frac{|\text{operaciones de punto flotante}|}{t \times 10^9} \tag{2.1}$$

siendo t el tiempo total de ejecución expresado en segundos.

Si bien es una métrica popular y ampliamente utilizada, GFLOPS puede resultar parcialmente útil para algunos problemas (p. ej. aquellos que son *memory-bound*) o directamente no tener utilidad en otros (p. ej. aquellos que computan en aritmética de enteros).

2.4.2. Portabilidad

La evaluación de la portabilidad de rendimiento es esencial para asegurar que una aplicación pueda aprovechar al máximo los recursos disponibles en diversas plataformas, mantener un rendimiento óptimo y adaptarse a los cambios tecnológicos en el campo de HPC. Resulta fundamental contar con métricas objetivas y métodos de evaluación sólidos que permitan hacer comparaciones justas y consistentes entre distintas aplicaciones y sistemas. Sin embargo, la falta de consenso en cuanto a su definición precisa y la forma de medir dicho rendimiento da lugar a conclusiones subjetivas y, en muchos casos, a resultados contradictorios en la literatura. Esto supone un desafío significativo para los arquitectos de software y hardware, especialmente en el ámbito de HPC, donde las aplicaciones deben ejecutarse de manera eficiente en plataformas heterogéneas.

La evaluación de la portabilidad se puede dividir en dos aspectos principales: la portabilidad funcional y la portabilidad de rendimiento [101, 102].

- La portabilidad funcional se refiere a la capacidad de una aplicación de ejecutarse correctamente en diferentes plataformas, independientemente de las diferencias en la arquitectura del hardware y del sistema operativo. En otras palabras, una aplicación funcionalmente portable es aquella que puede ejecutarse de manera consistente y sin errores en diferentes entornos. Para evaluar la portabilidad funcional, se realizan pruebas exhaustivas en diferentes plataformas para identificar y resolver problemas de compatibilidad y dependencias específicas del sistema.
- Por otro lado, la portabilidad de rendimiento se refiere a la capacidad de una aplicación de mantener un rendimiento óptimo en diferentes plataformas. Esto implica que la aplicación pueda aprovechar eficientemente los recursos disponibles en cada plataforma, como la capacidad de cálculo, la memoria y la interconexión de red. La evaluación de la portabilidad de rendimiento implica medir y comparar el rendimiento de una aplicación en diferentes configuraciones de hardware y software, identificando posibles cuellos de botella y optimizando el código para maximizar el rendimiento en cada plataforma.

La portabilidad de rendimiento en una aplicación está estrechamente relacionada con la eficiencia tanto de la aplicación en sí como de la arquitectura en la que se ejecuta. Es por esto que se desprenden dos conceptos:

- Eficiencia de aplicación: hace referencia al rendimiento logrado en una plataforma determinada, estandarizado en relación con el mejor rendimiento conocido de la implementación de una aplicación en esa misma plataforma. Es decir, esta medida analiza el rendimiento de una aplicación en comparación con la implementación más rápida conocida en esa plataforma específica.
- Eficiencia arquitectónica: mide el rendimiento logrado de una aplicación en una plataforma dada, normalizado con respecto al máximo rendimiento teórico o práctico que puede ser alcanzado en dicha plataforma. Esta métrica evalúa hasta qué punto una aplicación utiliza los recursos de la plataforma en la que está implementada, en relación con dos niveles de referencia de rendimiento: el pico teórico y el pico práctico alcanzable mediante la optimización de todos los recursos de la plataforma.

De acuerdo con Penycook *et al.* [103], la portabilidad de rendimiento se refiere a *"Una medida de la eficiencia de rendimiento de una aplicación para un problema dado que puede ejecutarse correctamente en todas las plataformas de un conjunto dado"*. Estos autores definen dos métricas diferentes de eficiencia de rendimiento, una para la eficiencia de aplicación y otra para la eficiencia arquitectónica.

Los autores definen la métrica de portabilidad de rendimiento como la media armónica de la eficiencia de rendimiento de una aplicación observada en un conjunto de plataformas. Si la aplicación falla en alguna(s) plataforma(s) medida(s), entonces se define la portabilidad de rendimiento como 0. Formalmente, para un conjunto dado de plataformas H de la misma clase de arquitectura, la portabilidad de rendimiento Φ de una aplicación de caso de estudio α que resuelve el problema p es:

$$\Phi(\alpha, p, H) = \begin{cases} \frac{|H|}{\sum_{i \in H} \frac{1}{e_i(\alpha,p)}} & \text{si } i \text{ es compatible } \forall i \in H \\ 0 & \text{en caso contrario} \end{cases} \tag{2.2}$$

donde $e_i(\alpha, p)$ corresponde a la eficiencia de rendimiento de la aplicación de estudio α que resuelve el problema p en la plataforma i.

Sin embargo, Marowka [104] identificó las siguientes limitaciones en la métrica propuesta por Penycook:

- La métrica es poco intuitiva, no familiar, pierde información, es difícil de usar y los puntajes de portabilidad de rendimiento que produce son irreales. Por ejemplo, si una plataforma no soporta una aplicación, sugiere que la portabilidad de rendimiento de la aplicación es cero, lo cual no tiene sentido.
- La media armónica utilizada sigue los valores bajos de rendimiento, incluso cuando los valores de otras plataformas son significativamente más altos. Esto no refleja adecuadamente mejoras en sistemas híbridos.
- La métrica es muy sensible al tamaño del problema. Pequeñas variaciones en el tamaño de entrada pueden producir grandes cambios en la puntuación.
- Diferentes estudios han utilizado esta métrica de maneras inconsistentes; por ejemplo, sólo considerando las plataformas que soportan la aplicación para evitar puntajes de cero. Esto lleva a resultados no uniformes y difícilmente reproducibles.
- No hay consenso sobre qué enfoque de eficiencia de rendimiento (por aplicación o arquitectura) se debe utilizar para calcular la portabilidad de rendimiento. Cada uno produce diferentes resultados y no está claro cuál es mejor.
- No existen pautas claras sobre cómo medir y reportar los resultados de manera que sean justos, comparables y significativos. Por ello es difícil sacar conclusiones bien fundadas de los distintos estudios.

Es por esto que Marowka posteriormente reformula la métrica de portabilidad de rendimiento de la siguiente manera:

$$\bar{\Phi}(\alpha, p, H) = \begin{cases} \frac{\sum_{i \in H} e_i(\alpha,p)}{|H|} & \text{si } i \text{ es compatible } \forall i \in H \\ \text{no aplicable (NA)} & \text{en caso contrario} \end{cases} \tag{2.3}$$

En base a esta nueva fórmula, se pueden destacar las siguientes mejoras:

- Definición y objetivos claros: Marowka propone un enfoque estandarizado y objetivo basado en reglas y directrices estrictas para la medición y el reporte del rendimiento. Esto contrasta con la métrica de Penycook, donde diferentes estudios han aplicado variaciones y han encontrado dificultades para obtener resultados coherentes.

- Métrica reformulada: la métrica definida por Marowka se basa en el promedio aritmético, en lugar del promedio armónico utilizado por Penycook. Esto resuelve problemas asociados con esta última, como la poca intuición, la pérdida de información, y resultados poco realistas.
- Enfoque en la eficiencia de rendimiento: la métrica de Marowka se basa en la eficiencia de rendimiento, considerando tanto la eficiencia de la aplicación como la eficiencia arquitectónica. Esto permite una evaluación más completa de la portabilidad del rendimiento desde diferentes perspectivas.
- Adopción de métodos de mediciones estándar: Marowka enfatiza la necesidad de adoptar métodos de medición estandarizados para la eficiencia de rendimiento, lo que facilita comparaciones más justas entre diferentes estudios y plataformas.

El concepto de portabilidad de rendimiento resalta la capacidad de escribir código que pueda utilizar eficientemente los recursos informáticos disponibles, como CPUs, GPUs o aceleradores especializados, manteniendo un determinado nivel de rendimiento independientemente de la configuración de hardware específica. Con la portabilidad de rendimiento, los desarrolladores pueden escribir código una única vez y obtener un rendimiento similar en distintas plataformas objetivo, sin la necesidad de realizar optimizaciones de código manuales o modificaciones específicas de la plataforma, lo que reduce el tiempo y el esfuerzo de desarrollo.

2.4.2.1. Ejemplo

Con el fin de ilustrar la aplicación de la métrica de portabilidad de rendimiento, se proporciona la Tabla 2.1 que muestra un ejemplo de cálculo de esta métrica, considerando tanto la eficiencia arquitectónica, que compara el rendimiento logrado con el rendimiento pico teórico de la plataforma, como la eficiencia de la aplicación, que evalúa el rendimiento logrado respecto al rendimiento óptimo alcanzado por una implementación de referencia bien optimizada.

Para ejemplificar, se considera un conjunto de plataformas *A, B, C, D, E*. La Tabla 2.2 ilustra cómo la eficiencia tanto arquitectónica como de aplicación varía al excluir plataformas sin implementación soportada (como la *Plataforma D*) o con rendimiento poco óptimo (como la *Plataforma C*). Por ejemplo, al considerar solo las plataformas *A, B*, se obtiene una eficiencia arquitectónica del 82.50 % y una eficiencia de aplicación del 86.07 %, reflejando la alta portabilidad entre estas dos plataformas. Cuando se agrega la *Plataforma D*, al no ser una plataforma soportada por la aplicación, no es posible calcular las eficiencias, por más que en el conjunto haya plataformas soportadas. Estos cálculos son fundamentales para la toma de decisiones en el diseño y optimización de software para múltiples arquitecturas.

2.4.3. Costo de programación

Medir el costo o esfuerzo de programación en las aplicaciones es un desafío constante. La programación es una actividad altamente compleja que involucra la creación y mantenimiento de código fuente, y determinar con precisión el esfuerzo requerido para llevar a cabo estas tareas puede resultar complejo.

La métrica SLOC es ampliamente utilizada para medir el tamaño de un programa y, por ende, como una estimación del costo o esfuerzo requerido para desarrollarlo. Esta métrica cuenta el número de líneas de código fuente escritas por los programadores. Sin embargo, es

Plataforma	GFLOPS Logrados	GFLOPS Pico	GFLOPS Mejor	Eficiencia Arq.	Eficiencia Aplic.
	200	250	235	80 %	85 %
B	595	700	683	85 %	87.14 %
C	40	500	333	8 %	12 %
D	N/A	800	N/A	N/A	N/A
E	125	1250	868	10 %	14.40 %

Tabla 2.1: Ejemplo de portabilidad de rendimiento: plataformas utilizadas.

Conjunto de Plataformas	Eficiencia Arq.	Eficiencia Aplic.
A, B, C, D, E	N/A	N/A
A, B, C, E	45.75 %	49.64 %
A, B, C	57.66 %	61.38 %
A, B	82.50 %	86.07 %
A	80.00 %	85.00 %

Tabla 2.2: Ejemplo de portabilidad de rendimiento en 5 plataformas ficticias.

importante tener en cuenta que SLOC no es una medida perfecta y presenta tanto ventajas como desventajas.

Por un lado, una de las principales ventajas de SLOC es su simplicidad y facilidad de aplicación. El conteo de líneas de código fuente es una tarea relativamente sencilla y puede ser automatizada utilizando herramientas específicas. Además, SLOC proporciona una medida objetiva que puede ser utilizada para comparar diferentes programas o versiones del mismo. A su vez, SLOC tiene la capacidad de proporcionar una estimación inicial del esfuerzo requerido para desarrollar un programa. Al contar las líneas de código, se puede obtener una idea aproximada de la cantidad de trabajo involucrado [105].

Por otro lado, una de las principales limitaciones es que no todas las líneas de código son iguales. Es por esto que la complejidad de un programa no se puede medir únicamente por el número de líneas de código, debido a que algunas de estas pueden ser simples y directas, mientras que otras pueden ser más complejas y requerir un mayor esfuerzo para su implementación. Además, SLOC no tiene en cuenta otros elementos importantes del código fuente, como los comentarios, los espacios en blanco o el formateo particular. Estos elementos pueden influir en la legibilidad y mantenibilidad del código, pero no se reflejan en el conteo de líneas del mismo. Por lo tanto, SLOC no proporciona una medida completa de la complejidad o el esfuerzo de programación involucrado, pero sirve como una base para comenzar a comprender estos aspectos [106].

2.5. Bioinformática

En esta sección, se aborda los conceptos fundamentales del campo de la bioinformática, enfocándose en el ASB y los algoritmos asociados. La Sección 2.5.1 introduce el tema, des-

tacando sus aplicaciones prácticas. Posteriormente, en la Sección 2.5.2, se profundiza en el ASB, una herramienta fundamental en la bioinformática. La Sección 2.5.3 se dedica a explorar los diversos algoritmos utilizados para el ASB, seguida por la Sección 2.5.4, que clasifica estos algoritmos en categorías como alineamientos globales y locales, y métodos basados en programación dinámica y heurísticas. La Sección 2.5.5 se centra en los algoritmos basados en programación dinámica, detallando enfoques globales, locales, semi-globales y solapados. El capítulo continúa con la Sección 2.5.6, que presenta las bases de datos biológicas más relevantes, incluyendo colaboraciones internacionales y bases de datos específicas de proteínas y nucleótidos. Finalmente, las secciones 2.5.7 y 2.5.8 abordan la aceleración del ASB y GCUPS, respectivamente, culminando con la Sección 2.5.9, que discute las implementaciones para procesamiento en GPU, resaltando su importancia en la optimización de procesos bioinformáticos. Este capítulo provee un análisis integral de los métodos y herramientas clave en bioinformática, enfatizando su aplicación y desarrollo tecnológico.

2.5.1. Introducción

La bioinformática es un campo interdisciplinario que combina la biología, la informática y la tecnología de la información para analizar e interpretar datos biológicos. Tiene sus orígenes a mediados del siglo 20, cuando la llegada de las computadoras y la elucidación de la estructura molecular del ADN sentaron las bases para el uso de métodos computacionales en la investigación biológica.

En la década de 1960 se vieron los primeros avances en la integración de herramientas computacionales en la biología molecular, a partir del desarrollo del primer ensamblador de secuencias de péptidos de novo, la base de datos de secuencias de proteínas y el modelo de sustitución de aminoácidos para la filogenética. En estos años también se vio la formalización del código genético, que estableció al ADN como el portador primario de información biológica [107].

Durante las siguientes dos décadas, se experimentaron avances notables tanto en biología molecular como en informática, lo cual tuvo un impacto fundamental en el análisis de genomas completos. Además, se logró una mayor facilidad en el almacenamiento y recuperación de datos biológicos, lo cual impulsó aún más el desarrollo de este campo.

El uso extendido de internet y la introducción de la secuenciación de próxima generación en las décadas de 1990 y 2000, generaron un incremento significativo en la cantidad de datos biológicos y una rápida proliferación de herramientas de bioinformática, marcando un punto de inflexión en esta área de estudio.

En la actualidad, la bioinformática se enfrenta a múltiples desafíos, tanto por la necesidad de manejar grandes cantidades de datos, como también para garantizar la reproducibilidad de los resultados. La bioinformática continúa en constante evolución, con un enfoque cada vez mayor en la biología de sistemas y la modelización computacional de organismos completos y sus entornos.

2.5.1.1. Aplicaciones

Entre las aplicaciones bioinformáticas más destacadas, se pueden nombrar:

- Análisis de secuencias genéticas y proteicas: es una de las aplicaciones más esenciales que se enfoca en el análisis de secuencias de ADN, ARN y proteínas en diversas herramientas bioinformáticas para identificar genes, comparar secuencias y predecir la estructura y función de las proteínas [108].

- Genómica y transcriptómica: son áreas de estudio en las que la bioinformática desempeña un papel fundamental al facilitar el análisis de genomas completos y la expresión génica, lo que implica el ensamblaje del genoma, la anotación genómica y el análisis de datos de secuenciación de próxima generación [109].
- Proteómica: es otro campo en el que la bioinformática juega un papel fundamental al permitir la identificación y caracterización de proteínas a gran escala, a través del uso de técnicas como la espectrometría de masas y la identificación de rutas metabólicas y de señalización [110].
- Metagenómica: se enfoca en el estudio de comunidades microbianas y la bioinformática desempeña un rol esencial en este campo, especialmente en el análisis de muestras ambientales o clínicas para estudiar la diversidad y función de los microorganismos [111].
- Farmacogenómica y diseño de fármacos: la bioinformática desempeña un papel importante en la identificación de blancos farmacológicos y en el diseño asistido por computadora de fármacos, incluyendo la modelización de la interacción fármaco-receptor [112].
- Biología de sistemas y redes biológicas: son áreas en las que la bioinformática es de gran utilidad, ya que ayuda en la construcción y análisis de redes biológicas, lo que permite entender mejor los sistemas biológicos complejos [113].
- Bioinformática en medicina personalizada: permite la genómica a nivel individuo, el diagnóstico basado en biomarcadores y la terapia dirigida, lo que contribuye a un enfoque más preciso y efectivo en el tratamiento de enfermedades [114].

2.5.2. Alineamiento de secuencias biológicas (ASB)

Originalmente, el ASB se desarrolló como una herramienta para entender las relaciones evolutivas y funcionales entre secuencias de proteínas y ácidos nucleicos, pero en la actualidad, se convirtió en una operación fundamental de la bioinformática y la biología molecular, abarcando desde el análisis filogenético [115], la detección de enfermedades genéticas [116], la identificación y cuantificación de regiones conservadas o unidades funcionales [117, 118], así como en la predicción y perfilado de secuencias ancestrales [119]. Además, su importancia se extiende al desarrollo de nuevos fármacos y a la investigación forense criminal, lo que ha llevado a la comunidad científica a realizar un gran esfuerzo en este campo de la bioinformática [120].

El ASB es una herramienta útil para identificar secuencias homólogas, es decir, secuencias que comparten un ancestro común. Estas pueden ser ortólogas, derivadas de un evento de especiación, o parálogas, resultantes de duplicaciones genómicas dentro de una misma especie. El reconocimiento de secuencias homólogas es esencial para inferir funciones de genes y proteínas desconocidas y para entender la historia evolutiva de los organismos [121]. En forma simplificada, el ASB permite determinar qué tan parecidas son dos secuencias biológicas.

2.5.3. Algoritmos para ASB

Las secuencias de ADN y proteínas son cadenas de moléculas construidas a partir de un conjunto de nucleótidos y aminoácidos, respectivamente. Por un lado, el alfabeto del ADN

consta de cuatro nucleótidos diferentes, representados por: adenina (A), timina (T), citosina (C) y guanina (G), formando la estructura de doble hélice del ADN, donde A siempre se empareja con T y C con G. Por otro lado, las proteínas utilizan un alfabeto más complejo de 20 aminoácidos diferentes, cada uno con propiedades únicas que determinan la estructura y función de la proteína [122].

Las mutaciones en estas secuencias pueden manifestarse como sustituciones, donde un símbolo es reemplazado por otro, inserciones, donde un nuevo símbolo en insertado en una secuencia y eliminaciones, donde un símbolo es eliminado de una secuencia. Esto representa cambios importantes debido a que pueden provocar variaciones significativas en la estructura y función de las moléculas biológicas, variando la longitud y composición de la secuencia [123].

Este desafío se aborda mediante la introducción de *gaps* (huecos) en las secuencias que compensan las diferencias de longitud causadas por inserciones y eliminaciones, permitiendo un alineamiento más preciso que refleje las relaciones evolutivas y funcionales entre las secuencias. La Figura 2.24 muestra un ejemplo de un alineamiento de dos secuencias de ADN sin *gap* y con *gap*.

En este contexto, la matriz de puntuación y sustitución son fundamentales.

Figura 2.24: Alineamiento simple entre dos secuencias de ADN.

Por un lado, la matriz de puntuación asigna valores a las correspondencias y discrepancias entre las secuencias, incluyendo la penalización por la introducción de *gaps*, las coincidencias (conocidas también en inglés por *matches*) y no coincidencias (conocidas también en inglés por *mismatches*). Estas puntuaciones ayudan a optimizar el alineamiento, equilibrando la necesidad de representar con precisión las relaciones evolutivas con la minimización de las distorsiones causadas por *gaps* excesivos.

Por otro lado, la matriz de sustitución ofrece una perspectiva sobre la evolución de las secuencias a través del tiempo. Esta matriz proporciona puntuaciones para las sustituciones de nucleótidos o aminoácidos, reflejando la probabilidad de cambios durante la evolución. Estas puntuaciones se basan en la identidad (cuando las unidades son idénticas) y la similitud (cuando las unidades difieren, pero son funcional o estructuralmente similares) entre las secuencias. En el caso de las secuencias de proteínas, es común emplear las familias de matrices PAM y BLOSUM, las cuales asignan puntuaciones a las sustituciones de aminoácidos basándose en su similitud evolutiva. La Figura 2.25 muestra un ejemplo de una de las matrices BLOSUM (BLOSUM62). Por otro lado, en el caso de las secuencias de ADN la situación es más sencilla, debido a que se suele utilizar un valor fijo para las coincidencias y otro para las no coincidencias [124].

2.5.4. Clasificación de algoritmos

Los algoritmos de alineamiento se clasifican según varios criterios, siendo los más relevantes el alcance del alineamiento (global o local) y el enfoque computacional empleado (programación dinámica o heurística). La Figura 2.26 ilustra esta clasificación.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X	*
A	4	-1	-2	-2	0	-1	-1	0	-2	-1	-1	-1	-1	-2	-1	1	0	-3	-2	0	-2	-1	0	-4
R	-1	5	0	-2	-3	1	0	-2	0	-3	-2	2	-1	-3	-2	-1	-1	-3	-2	-3	-1	0	-1	-4
N	-2	0	6	1	-3	0	0	0	1	-3	-3	0	-2	-3	-2	1	0	-4	-2	-3	3	0	-1	-4
D	-2	-2	1	6	-3	0	2	-1	-1	-3	-4	-1	-3	-3	-1	0	-1	-4	-3	-3	4	1	-1	-4
C	0	-3	-3	-3	9	-3	-4	-3	-3	-1	-1	-3	-1	-2	-3	-1	-1	-2	-2	-1	-3	-3	-2	-4
Q	-1	1	0	0	-3	5	2	-2	0	-3	-2	1	0	-3	-1	0	-1	-2	-1	-2	0	3	-1	-4
E	-1	0	0	2	-4	2	5	-2	0	-3	-3	1	-2	-3	-1	0	-1	-3	-2	-2	1	4	-1	-4
G	0	-2	0	-1	-3	-2	-2	6	-2	-4	-4	-2	-3	-3	-2	0	-2	-2	-3	-3	-1	-2	-1	-4
H	-2	0	1	-1	-3	0	0	-2	8	-3	-3	-1	-2	-1	-2	-1	-2	-2	2	-3	0	0	-1	-4
I	-1	-3	-3	-3	-1	-3	-3	-4	-3	4	2	-3	1	0	-3	-2	-1	-3	-1	3	-3	-3	-1	-4
L	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4	-2	2	0	-3	-2	-1	-2	-1	1	-4	-3	-1	-4
K	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5	-1	-3	-1	0	-1	-3	-2	-2	0	1	-1	-4
M	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5	0	-2	-1	-1	-1	-1	1	-3	-1	-1	-4
F	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6	-4	-2	-2	1	3	-1	-3	-3	-1	-4
P	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7	-1	-1	-4	-3	-2	-2	-1	-2	-4
S	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4	1	-3	-2	-2	0	0	0	-4
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5	-2	-2	0	-1	-1	0	-4
W	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11	2	-3	-4	-3	-2	-4
Y	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7	-1	-3	-2	-1	-4
V	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4	-3	-2	-1	-4
B	-2	-1	3	4	-3	0	1	-1	0	-3	-4	0	-3	-3	-2	0	-1	-4	-3	-3	4	1	-1	-4
Z	-1	0	0	1	-3	3	4	-2	0	-3	-3	1	-1	-3	-1	0	-1	-3	-2	-2	1	4	-1	-4
X	0	-1	-1	-1	-2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-2	0	0	-2	-1	-1	-1	-1	-1	-4
*	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	1

Figura 2.25: Matriz de sustitución BLOSUM62.

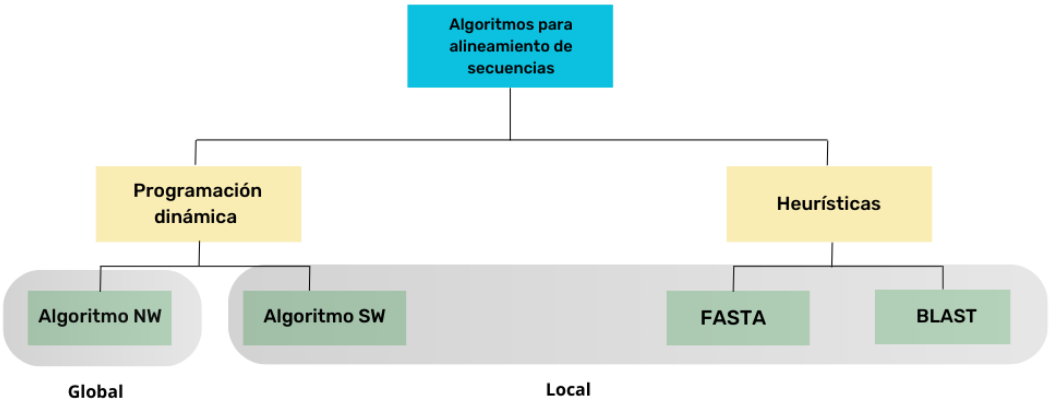

Figura 2.26: Clasificación de algoritmos para alineamiento de secuencias.

2.5.4.1. Alineamientos globales y locales

Por un lado, el alineamiento global es un método que compara dos secuencias en su totalidad, buscando la mejor correspondencia entre ellas desde el principio hasta el final, siendo un enfoque útil cuando las secuencias tienen longitudes similares y se espera que compartan una relación evolutiva cercana. Por otro lado, el alineamiento de secuencias local busca regiones de alta similitud entre las secuencias, sin considerar su longitud total, siendo más adecuado para secuencias que pueden tener regiones conservadas, pero no necesariamente una relación evolutiva cercana en su totalidad.

2.5.4.2. Programación dinámica y heurísticas

La programación dinámica es una técnica avanzada en el diseño de algoritmos que se utiliza ampliamente para resolver problemas complejos mediante la descomposición en subproblemas más manejables. En el contexto del ASB, la programación dinámica es utilizada para la construcción de la matriz de puntuación, en la que cada celda representa un subproblema, es decir, una correspondencia parcial entre segmentos de las secuencias. El algoritmo llena esta matriz y encuentra el camino que representa el mejor alineamiento. La principal ventaja de la programación dinámica es su precisión, debido a que garantiza encontrar el alineamiento óptimo basándose en un esquema de puntuación definido. No obstante, la clara desventaja que tiene es la necesidad de utilizar altos recursos computacionales, especialmente en términos de memoria y tiempo, lo cual se convierte en un problema cuando se trabajan con secuencias muy largas.

Por otro lado, los enfoques basados en heurísticas proporcionan una opción más eficiente en términos de recursos y tiempo, aunque no garantizan la obtención del alineamiento óptimo. Estos métodos aplican reglas o estrategias para alcanzar una solución aceptable en períodos de tiempo más cortos. Consiste primero en identificar las regiones de alta similitud y luego expandir estos alineamientos parciales, pero sin examinar todas las posibles formas de alinear las secuencias, sino que se enfocan en las más prometedoras. BLAST y FASTA son ejemplos representativos de algoritmos heurísticos utilizados para alinear secuencias y buscar similitudes en bases de datos de gran tamaño. Como se explicó, su mayor ventaja radica en la eficiencia, aunque estos enfoques pueden pasar por alto algunas correspondencias importantes, especialmente en casos de secuencias altamente divergentes o complejas.

2.5.5. Algoritmos basados en programación dinámica

Son dos los algoritmos de alineamiento basados en programación dinámica más conocidos: el algoritmo de Needleman-Wunsch (NW) para alineamiento global y el algoritmo de Smith-Waterman (SW) para alineamiento local. Además, existen alternativas que varían sus métodos, como el alineamiento Semi-global y el de Solapamiento. Estos algoritmos constan de tres etapas principales: (1) inicialización de la matriz, (2) relleno de la matriz, (3) retroceso.

2.5.5.1. Global

El algoritmo de NW es utilizado para alinear dos secuencias completas, desde el inicio hasta el final, optimizando el alineamiento a lo largo de toda la longitud de las secuencias. Este algoritmo es especialmente útil para secuencias de longitud similar y se espera que estén relacionadas en su totalidad. Para construir una matriz de puntuación se utiliza la programación dinámica, donde cada celda representa el puntaje óptimo del alineamiento hasta ese punto. Se consideran tres posibilidades: alineación de dos residuos, inserción de un *gap* en la primera secuencia o inserción de un *gap* en la segunda secuencia. Además, se aplican penalizaciones tanto para la apertura como para la extensión de *gaps*, lo que evita el alineamiento con un exceso de estos. Este algoritmo es ideal para comparar secuencias con alta homología y longitud similar y es una representación clara del alineamiento global.

Dadas dos secuencias Q y D de longitud $|Q| = m$ y $|D| = n$, el algoritmo NW construye una matriz de puntuación G de tamaño $(m + 1) \times (n + 1)$ para calcular la puntuación de alineamiento global. La matriz se inicializa y se rellena de acuerdo con las siguientes reglas:

- Inicialización: la primera fila y columna de la matriz G se inicializan con valores que

representan las penalizaciones acumuladas por los *gaps*. Esto se hace para representar la inserción de *gaps* al principio de las secuencias.

$$\begin{array}{ll} G_{i,0} = i \times G_o & \text{para } 0 \leq i \leq m \\ G_{0,j} = j \times G_o & \text{para } 0 \leq j \leq n \end{array} \tag{2.4}$$

Donde G_o es la penalización por la apertura de un *gap*.

- Relleno de la matriz: la matriz se rellena iterando sobre sus filas y columnas, calculando cada celda $G_{i,j}$ mediante la siguiente relación de recurrencia:

$$G_{i,j} = \text{máx} \begin{cases} G_{i-1,j-1} + SM(Q[i], D[j]) \\ G_{i-1,j} - G_o \\ G_{i,j-1} - G_o \end{cases} \tag{2.5}$$

  Donde $SM(Q[i], D[j])$ es la puntuación de la matriz de similitud para los residuos $Q[i]$ y $D[j]$ y G_o es la penalización por apertura de *gap*.

- Retroceso: una vez que la matriz está completamente llena, el alineamiento óptimo se obtiene mediante un proceso de retroceso, comenzando desde $G_{m,n}$, siguiendo la trayectoria que generó el puntaje máximo hasta llegar a $G_{0,0}$.

Ejemplo de NW: la Figura 2.27 muestra un ejemplo de alineamiento entre las secuencias de proteínas *ASASTGE* y *ATGSMPL* de acuerdo al método NW. En este caso, se emplea el esquema de puntuación que utiliza la matriz BLOSUM62, y las penalizaciones por *gap* se puntúan con 8. El puntaje óptimo obtenido es de 3.

2.5.5.2. Local

El algoritmo de SW es un método de alineamiento local diseñado para identificar regiones de alta similitud dentro de secuencias más largas y de diferente longitud. A diferencia del alineamiento global, este algoritmo no alinea las secuencias completas, sino que encuentra las subsecuencias que se alinean mejor. La metodología es similar al algoritmo NW en la construcción de una matriz de puntuación, pero se enfoca en identificar la subsecuencia con la puntuación más alta, comenzando y terminando donde la puntuación es máxima. Al igual que en el alineamiento global, se aplican penalizaciones por la apertura y extensión de *gaps*. Este algoritmo es utilizado cuando se sospecha que solo ciertas partes de las secuencias están conservadas o relacionadas [125].

A continuación, se describe el algoritmo SW:

- Inicialización: la inicialización se realiza adjudicando el valor cero a todos los elementos de la matriz de puntuación *H*. A diferencia del algoritmo NW, no se aplican penalizaciones en la primera fila y columna, ya que el alineamiento local puede comenzar en cualquier punto de las secuencias.

$$\begin{array}{ll} H_{i,0} = 0 & \text{para } 0 \leq i \leq m \\ H_{0,j} = 0 & \text{para } 0 \leq j \leq n \end{array} \tag{2.6}$$

  Esto asegura que el algoritmo tenga la flexibilidad de identificar la subsecuencia de mayor similitud sin estar restringido a alineamientos que comienzan al principio de las secuencias.

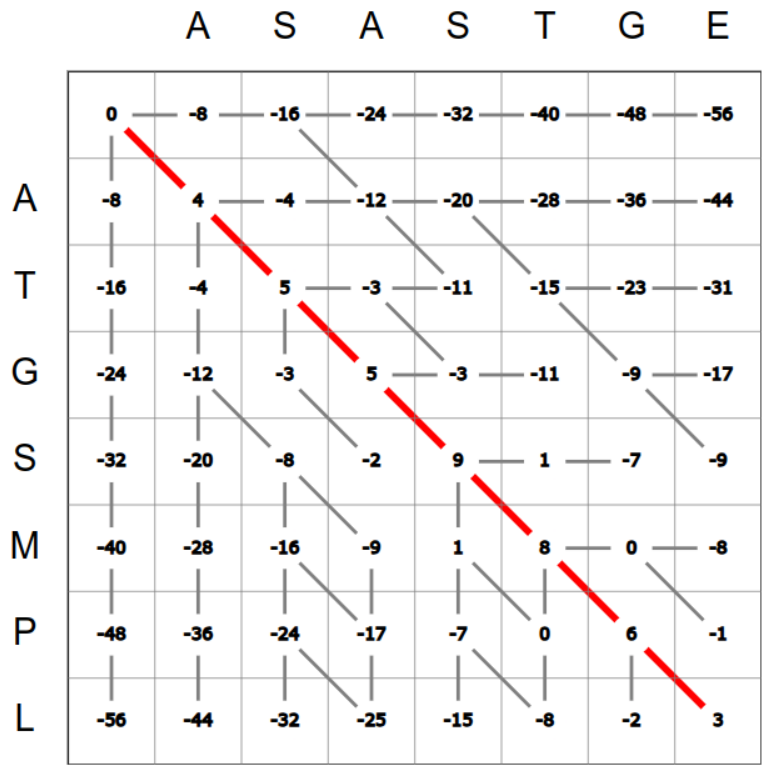

Figura 2.27: Ejemplo de alineamiento entre secuencias ASASTGE y ATGSMPL de acuerdo al método NW.

- Relleno de la matriz: el proceso de relleno de la matriz de puntuación H sigue las siguientes reglas, utilizando las modificaciones de Gotoh para admitir un modelo de penalización de *gap* por afinidad [126]:

$$H_{i,j} = max \begin{cases} 0 \\ H_{i-1,j-1} + SM(Q[i], D[j]) \\ E_{i,j} \\ F_{i,j} \end{cases} \tag{2.7}$$

$$E_{i,j} = max \begin{cases} H_{i,j-1} - G_o \\ E_{i,j-1} - G_e \end{cases} \tag{2.8}$$

$$F_{i,j} = max \begin{cases} H_{i-1,j} - G_o \\ F_{i-1,j} - G_e \end{cases} \tag{2.9}$$

Donde $SM(Q[i], D[j])$ es la puntuación de la matriz de similitud para los residuos $Q[i]$ y $D[j]$, G_o es la penalización por apertura de *gap* y G_e es la penalización por extensión de *gap*. La opción de asignar un valor de cero en las ecuaciones asegura que solo se consideren alineamientos con puntuaciones positivas, lo que es fundamental para el alineamiento local.

- Retroceso: el proceso de retroceso comienza desde el valor más alto en la matriz H, que representa el puntaje de alineamiento local óptimo. Desde este punto, se retrocede a través de la matriz siguiendo el camino que llevó a este puntaje máximo, hasta que se encuentra un valor de cero. Este proceso reconstruye el alineamiento local óptimo entre las subsecuencias.

Ejemplo de SW: la Figura 2.28 muestra un ejemplo de alineamiento entre las secuencias de proteínas *ASASTGE* y *ATGSMPL* de acuerdo al método SW. El esquema de puntuación empleado es el siguiente: se utiliza la matriz de puntuación BLOSUM62, mientras que las penalizaciones por inserción y extensión de *gap* se puntúan con 10 y 2, respectivamente. El puntaje óptimo es igual a 12.

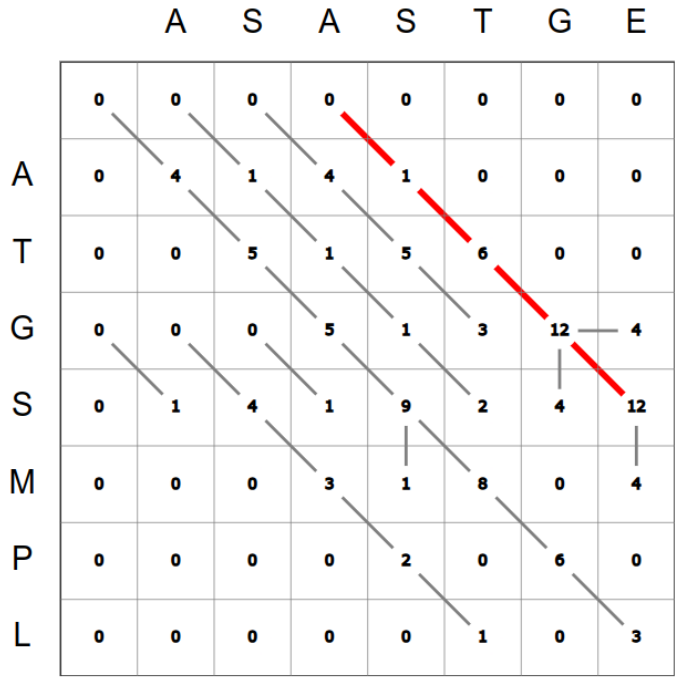

Figura 2.28: Ejemplo de alineamiento entre secuencias ASASTGE y ATGSMPL de acuerdo al método SW.

A diferencia de los alineamientos globales, los alineamientos locales toman en cuenta la similitud entre regiones pequeñas de las dos secuencias. Esto facilita la exposición de las similitudes entre secuencias iguales y, al mismo tiempo, proporciona resultados con un mayor sentido biológico. Por esta razón, los alineamientos globales son mucho menos utilizados que los locales [127].

2.5.5.3. Semi-global

El alineamiento semi-global es una variante que combina características de los alineamientos global y local. El objetivo es alinear secuencias en las que se espera que un extremo (ya sea el inicio o fin) esté bien alineado, mientras que el otro extremo puede no estarlo. La metodología utilizada es similar a los anteriores dos, pero con penalizaciones modificadas

para los espacios al inicio o al final de las secuencias. Este tipo de alineamiento es adecuado para situaciones en las que interesa alinear secuencias de lectura corta con un genoma de referencia, ya que se espera un buen alineamiento en la mayor parte de la secuencia, pero no necesariamente en los extremos [128].

2.5.5.4. Solapado

El alineamiento solapado es una técnica utilizada en el ensamblaje de secuencias, especialmente en la secuenciación de genomas, con el objetivo de encontrar la forma más adecuada de superponer dos secuencias. Esta metodología se centra en los extremos de las secuencias, buscando la mejor coincidencia de solapamiento y dejando de lado las partes que no se superponen [129].

2.5.6. Bases de datos biológicas

Las bases de datos biológicas consisten en librerías de información biológica, recopiladas a partir de experimentos científicos, literatura publicada, tecnología de experimentos de alto rendimiento y análisis computacional. Contienen información de áreas de investigación que incluyen genómica, proteómica, metabolómica, expresión génica de *microarrays* y filogenética. La información contenida en las bases de datos biológicas incluye la función de los genes, su estructura, localización (tanto celular como cromosómica), efectos clínicos de las mutaciones, así como similitudes de secuencias y estructuras biológicas.

Las bases de datos biológicas se clasifican en tres categorías principales: secuencias, estructuras y funcionales

- Las secuencias de ácidos nucleicos y proteínas se almacenan en bases de datos de secuencias.
- Las bases de datos de estructuras almacenan las estructuras resueltas de ARN y proteínas.
- Las bases de datos funcionales proporcionan información sobre el papel fisiológico de los productos génicos, como actividades enzimáticas, fenotipos mutantes o vías biológicas.

2.5.6.1. Colaboración Internacional de Bases de Datos de Secuencias de Nucleótidos

El Proyecto de *Colaboración Internacional de Bases de Datos de Secuencias de Nucleótidos* (INSDC, por sus siglas en inglés) [130] es una iniciativa conjunta entre los tres principales proveedores de bases de datos biológicas a nivel mundial: el NCBI [131], el Banco de Datos de ADN de Japón (DDBJ, por sus siglas en inglés) [132] y el Laboratorio de Biología Molecular Europeo (EMBL, por sus siglas en inglés) [133]. Su objetivo principal es garantizar que toda la información relevante esté disponible para todos los usuarios, por lo que se lleva a cabo un intercambio diario de datos entre estas bases de datos. A continuación, se describen las bases de datos más populares.

2.5.6.2. Bases de datos de proteínas

Entre las bases de datos de proteínas se pueden destacar:

- *Protein Information Resource* (PIR): el PIR, ubicado en el Centro Médico de la Universidad de Georgetown, es un recurso público de bioinformática integrado que brinda apoyo a la investigación genómica y proteómica, así como a los estudios científicos. Dispone de bases de datos de secuencias de proteínas. Se trata de la primera colección exhaustiva de secuencias macromoleculares en el Atlas de Secuencia y Estructura de Proteínas, publicado entre 1964 y 1974 bajo la dirección de Margaret Dayhoff.

- UniprotKB: es una base de datos de acceso libre de secuencias de proteínas e información funcional, muchas de las cuales provienen de proyectos de secuenciación del genoma. Contiene una gran cantidad de información sobre la función biológica de las proteínas derivada de la literatura de investigación. Es mantenida por el consorcio UniProt, el cual está compuesto por varias organizaciones europeas de bioinformática y una fundación de Washington, DC, Estados Unidos.

    - Swiss-Prot: es una base de datos central en UniProtKB, conocida por contener secuencias de proteínas no redundantes y anotadas de forma manual. Esta base de datos se destaca por su combinación de información obtenida de la literatura científica y análisis computacionales, los cuales son minuciosamente evaluados por biocuradores. Su principal objetivo es proporcionar información completa y actualizada sobre proteínas específicas. Con el fin de garantizar que las anotaciones estén al día con los últimos avances científicos, se lleva a cabo una revisión y actualización regular. Cada entrada en UniProtKB/Swiss-Prot implica un análisis detallado de la secuencia de la proteína y la correspondiente literatura. Un aspecto distintivo es la fusión de secuencias del mismo gen y especie en una única entrada, lo que permite identificar y documentar cualquier diferencia entre estas secuencias. Estas diferencias pueden ser el resultado de variaciones como el empalme alternativo, variaciones naturales, inicios de secuencias incorrectos, límites de exones equivocados, cambios en el marco de lectura y conflictos no resueltos.
    - TrEMBL: la base de datos TrEMBL contiene registros computacionalmente analizados de alta calidad, los cuales están enriquecidos con anotaciones automáticas y es también un núcleo de UniProtKB. Fue introducida como respuesta al aumento del flujo de datos resultante de los proyectos genómicos, ya que el proceso de anotación manual de UniProtKB/Swiss-Prot, que consume tiempo y esfuerzo humano, no podía ser ampliado para incluir todas las secuencias de proteínas disponibles. Las traducciones de las secuencias de codificación anotadas en la base de datos de secuencias nucleotídicas EMBLBank/GenBank/DDBJ son procesadas automáticamente e ingresadas en UniProtKB/TrEMBL, quien también contiene secuencias de PDB y de predicción de genes, incluyendo Ensembl, RefSeq y CCDS.

- NR y Environmental NR: la base de datos de proteínas NR es una colección exhaustiva de secuencias de proteínas que están organizadas de manera no redundante, lo que significa que las secuencias de proteínas idénticas están representadas por un único número de acceso, disminuyendo significativamente la redundancia en la base de datos. La base de datos NR se utiliza en las búsquedas de BLAST de proteína-proteína, una herramienta que encuentra regiones de similitud local entre secuencias. Esta base de datos forma parte del proyecto RefSeq más grande en NCBI, dentro de los Institutos Nacionales de Salud de Estados Unidos.

2.5.6.3. Bases de datos de nucleicos

Entre las bases de datos de nucleicos se pueden destacar:

- DDBJ: el DDBJ es una base de datos biológica que recopila secuencias de ADN. DDBJ comenzó sus actividades de banco de datos en 1986 y sigue siendo el único de secuencias de nucleótidos en Asia. Aunque DDBJ recibe principalmente sus datos de investigadores japoneses, puede aceptar datos de colaboradores de cualquier otro país.
- EMBL: el EMBL es una institución de investigación en biología molecular que fue fundada en 1974 como una organización intergubernamental financiada por los estados miembros con fondos públicos destinados a la investigación. En este centro, se lleva a cabo una amplia gama de investigaciones en biología molecular a través de unos 85 grupos independientes.
- GenBank: es una colección de acceso abierto de todas las secuencias de nucleótidos disponibles públicamente y sus traducciones de proteínas. Es producida y mantenida por el NCBI, como parte INSDC. GenBank y sus colaboradores reciben secuencias producidas en laboratorios de todo el mundo, provenientes de más de cien mil organismos distintos. GenBank se ha convertido en una base de datos importante para la investigación en campos biológicos y ha crecido en los últimos años a un ritmo exponencial, duplicándose aproximadamente cada 18 meses.

2.5.7. Aceleración de ASB

El ASB es un proceso computacionalmente intensivo que enfrenta varios desafíos, entre los que se incluyen la gestión de grandes bases de datos de secuencias, la necesidad de precisión en el alineamiento y la demanda de recursos computacionales elevados. Además, las secuencias biológicas presentan variabilidad y complejidad que dificultan la identificación de alineamientos óptimos, proceso que se complica aún más con el crecimiento exponencial de los datos genómicos disponibles [134].

El cálculo de la matriz de similitud es la parte más costosa computacionalmente del algoritmo SW. Es importante destacar que las celdas de la matriz H no pueden ser computadas en cualquier orden debido a las dependencias de datos inherentes al problema. Para calcular el valor de una celda, es necesario haber calculado previamente tres valores: el de la celda vecina superior, el de la celda vecina izquierda y el de la celda vecina superior-izquierda, tal como se ilustra en la Figura 2.29.

El algoritmo SW se puede utilizar para calcular: (a) alineamientos de pares (uno a uno); generalmente asociadas con secuencias largas de ADN; o (b) búsquedas de similitud en bases de datos (uno a muchos), generalmente asociadas con el alineamiento de secuencias de proteínas. Aunque la naturaleza de procesamiento del algoritmo SW con las dependencias de datos en el cálculo $H_{i,j}$ es muy desafiante desde el punto de vista de la explotación del paralelismo, ambos enfoques fueron estudiados en la literatura aprovechando las capacidades SIMD [26] y se ilustran en la Figura 2.30:

- Paralelismo intra-tarea: se centra en la aceleración del cálculo de un solo alineamiento a la vez. Las implementaciones que adoptan este esquema suelen realizar el cálculo de las celdas de cada antidiagonal de la matriz en paralelo, aprovechando que estos cálculos son independientes entre sí. También es factible calcular varias celdas de una fila o columna simultáneamente; sin embargo, este enfoque no considera las dependencias

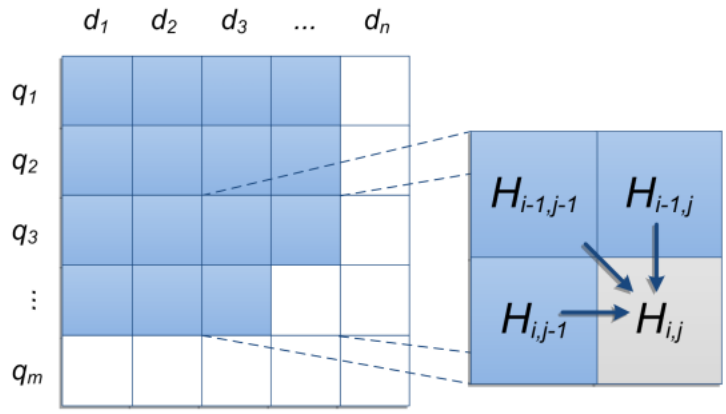

Figura 2.29: Dependencias de datos en la matriz H (extraído de [6]).

de datos mencionadas anteriormente, lo que requiere un ajuste posterior de los valores de las celdas para mantener la corrección del algoritmo. Este esquema es útil para resolver el caso (a).

- Paralelismo inter-tarea: se fundamenta en agilizar el procesamiento de múltiples alineamientos simultáneamente. La principal ventaja de este enfoque radica en la autonomía en el cálculo de cada alineamiento. Es útil para resolver el caso (b).

2.5.8. GCUPS

GCUPS es una medida de rendimiento especializada que se emplea en el ámbito de la bioinformática; en particular, para el ASB. Esta medida resulta relevante para las aplicaciones que llevan a cabo operaciones en matrices o espacios de celdas, como suele ser habitual en ASB, simulaciones de dinámica de fluidos y otros análisis computacionales intensivos.

Por definición, un CUPS representa la cantidad de tiempo necesario para computar una celda completa de la matriz de similitud H, incluyendo los cálculos relacionados con los valores en los arreglos E y F. Considerando una secuencia de consulta Q y una base de datos D, el rendimiento en GCUPS se calcula mediante la fórmula:

$$GCUPS = \frac{|Q| \times |D|}{t \times 10^9} \tag{2.10}$$

donde $|Q|$ denota el número total de símbolos en la secuencia de consulta, $|D|$ representa el número total de símbolos en la base de datos, y t es el tiempo total de ejecución expresado en segundos.

2.5.9. Implementaciones para GPU

Existen múltiples implementaciones en GPU para el ASB. Incluso, previo a la aparición de las GPUPUs, ya se habían realizado esfuerzos notables para acelerar el software mediante el uso de estos dispositivos. Debido a lo explicado en la sección 2.5.7, se presentan de forma

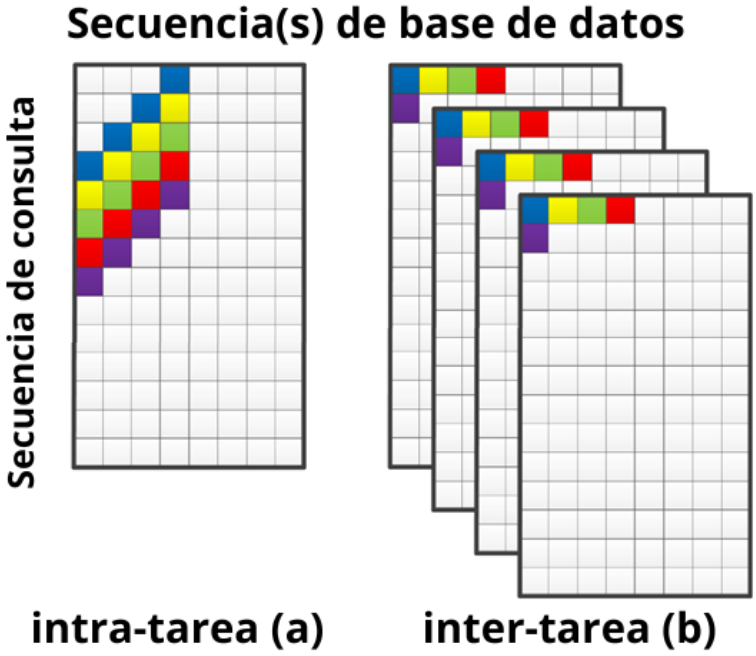

Figura 2.30: Enfoques de paralelización en cálculos de matrices de similitud (adaptado de [135]). Cada color indica las celdas que pueden ser calculadas juntas de manera SIMD.

separada las implementaciones de búsqueda de similitud (resumen en la Tabla 2.3) y de a pares (resumen en la Tabla 2.4).

2.5.9.1. Búsqueda de similitud

Inicialmente, en 2006, surgieron las primeras implementaciones en GPU gracias a los esfuerzos de Weiguo Liu *et al.* [136], así como de Yang Liu *et al.* [137]. Ambas propuestas, a pesar de sus similitudes como el uso de la librería OpenGL y el almacenamiento de secuencias en memoria de textura, presentaron características distintivas. Por ejemplo, la implementación de Weiguo Liu se limitaba a procesar secuencias proteicas de no más de 4096 aminoácidos, una restricción impuesta por la memoria de textura de la época. Esta limitación llevó a utilizar una versión reducida de la base de datos Swiss-Prot, logrando una velocidad de 0.65 GCUPS con una GPU NVIDIA GeForce 6800 GT. Por su parte, la propuesta de Yang Liu no tenía restricciones en cuanto a la longitud de las secuencias y ofrecía dos modos de ejecución, alcanzando entre 0.18 y 0.24 GCUPS con una GPU NVIDIA GeForce 7800 GTX.

En 2008, Manavski y Valle [138] presentaron la primera implementación utilizando CUDA para la búsqueda en bases de datos de proteínas SW, conocida como SW-CUDA. Esta propuesta se distinguió de las anteriores por adoptar un esquema de paralelismo inter-tareas, permitiendo a cada hilo de CUDA computar un alineamiento completo entre la secuencia de consulta y una secuencia particular de la base de datos. Además, SW-CUDA utilizó la técnica *Query Prfoile* (QP) para obtener puntuaciones de la matriz de sustitución, alcanzando 1.85

GCUPS en una GPU NVIDIA GeForce 8800 GTX y mostrando una buena escalabilidad con el número de GPUs.

En 2009, Yongchao Liu *et al.*[38] introdujeron CUDASW++, una implementación para GPUs con CUDA que combinaba los enfoques de paralelismo intra e inter-tareas. Esta herramienta estableció un umbral configurable de longitud para computar los alineamientos, optimizando el acceso a la memoria y explotando la jerarquía de la misma, lo que permitió alcanzar hasta 16.09 GCUPS en una configuración de doble GPU NVIDIA GeForce GTX 295.

Posteriormente, en 2010, los mismos autores presentaron una versión mejorada, CUDASW++ 2.0 [39], que ofrecía dos modos de ejecución e integraba la técnica QP y la aproximación de Farrar a través de la virtualización de instrucciones SIMD en GPUs. Esta versión alcanzó hasta 29.6 GCUPS en una configuración de doble GPU.

GASW, presentado también en 2010 [134], es otro software para GPUs compatibles con CUDA, destacándose por la eliminación de cuellos de botella en memoria y la conversión de la base de datos a un formato adecuado para el uso en GPU, logrando hasta 21.36 GCUPS en una GPU NVIDIA GeForce GTX 275.

Zou *et al.* [139] en 2011 propusieron una implementación basada en CUDA que combinaba diferentes optimizaciones como accesos globales a la memoria de manera coalescente, explotación de la jerarquía de memoria y desenrollado de bucles, alcanzando 28.35 GCUPS en una NVIDIA GeForce GTX 470.

Khalafallah *et al.* [140] en 2011 y Borovska y Lazarova [141] en 2012 utilizaron OpenCL y siguieron un enfoque inter-tareas, reutilizando varias optimizaciones de implementaciones anteriores y alcanzando hasta 65.99 GCUPS en diferentes configuraciones de hardware.

Dos años después, en 2013, Yongchao Liu *et al.* [40] presentaron CUDASW++ 3.0, dirigido a GPUs NVIDIA basadas en la arquitectura Kepler y combinando cálculos concurrentes de CPU y GPU. Esta versión, a pesar de las limitaciones en arquitecturas posteriores, se consideraba una de las implementaciones más rápidas para sistemas compatibles con CUDA, alcanzando 119 GCUPS con un procesador Intel i7 2700k y una GPU NVIDIA GeForce GTX 680.

En el año 2015, Matija Korpar *et al.* [37] presentaron SW#db, una herramienta de búsqueda de similitudes en bases de datos de secuencias, acelerada por GPU. SW#db emplea algoritmos de programación dinámica como SW, optimizados para CUDA y arquitecturas SIMD de CPU. La herramienta está diseñada para una búsqueda eficiente de similitudes exactas en bases de datos reducidas, utilizando tanto GPU como paralelización en CPU. Resultó ser significativamente más rápida que herramientas como CUDASW++, ofreciendo velocidades entre 4 a 5 veces mayores.

En el año 2017, Lan *et al.* presentan SWHybrid [142], un enfoque de trabajo híbrido para la búsqueda de secuencias de proteínas a gran escala en entornos de computación heterogéneos. Este marco integra unidades de procesamiento como CPUs, GPUs y Xeon Phis, abstrayéndolas como unidades de ejecución vectorial SIMD. SWHybrid utiliza una interfaz de programación unificada en C++ para abstraer diferencias arquitectónicas subyacentes, lo que facilita la optimización tanto de partes específicas de la arquitectura como de partes independientes de la arquitectura de la aplicación. SWhybrid alcanza aproximadamente 220 GCUPS y 250 GCUPS usando las GPUs NVIDIA Titan X y GTX1080, respectivamente.

En 2018, Sven Warris *et al.* presentaron pyPaSWAS [143], una herramienta basada en Python para el ASB utilizando múltiples núcleos de CPU y GPU. Esta herramienta es una implementación general de SW que se ejecuta en varias plataformas de hardware, proporcionando alineamientos de secuencias precisas y detalladas. La integración de Python con lenguajes de computación paralela de alto rendimiento en pyPaSWAS facilita su uso y ex-

tensión en bioinformática. La configuración más rápida utilizó una GPU NVIDIA GTX 1070 y ejecutó la implementación CUDA del algoritmo, resultando ser 2.8 veces más rápida que la versión OpenCL optimizada para GPU, alcanzando un rendimiento de 4.64 GCUPS.

En 2023, Bertil Schmidt *et al.* presentaron CUDASW++4.0 [41], una herramienta de software avanzada para el escaneo de bases de datos de secuencias proteicas utilizando el algoritmo de SW en GPUs compatibles con CUDA. Esta versión logra una alta eficiencia en el cálculo de alineamientos basadas en programación dinámica, minimizando el acceso a la memoria y las instrucciones ejecutadas. CUDASW++4.0 incorpora esquemas eficientes de teselado de matrices y partición de bases de datos de secuencias, y aprovecha la aritmética de punto flotante de próxima generación e instrucciones DPX. Esto resulta en un rendimiento cercano al pico en generaciones modernas de GPUs (Ampere, Ada, Hopper), con tasas de rendimiento de hasta 1.94 TCUPS, 5.01 TCUPS y 5.71 TCUPS en GPUs A100, L40S y H100, respectivamente.

2.5.9.2. Alineamiento de a pares

En el año 2010, Sandes y Melo presentaron la herramienta CUDAlign 1.0 [31], la cual se utiliza para alinear secuencias de ADN. Esta herramienta utiliza el método de *wavefront* para calcular la matriz de programación dinámica, asignando bloques a hilos en forma de paralelogramo, y proporciona como resultado el puntaje óptimo. Con el fin de mejorar el rendimiento, esta versión organiza cuidadosamente los datos en la jerarquía de memoria de la GPU. Al comparar secuencias de hasta 32 *Megabase Pairs* (MBP) en la GPU NVIDIA GTX 280, se obtuvo un máximo de 20.37 GCUPS.

En el año 2012, presentaron la versión CUDAlign 2.0 [32], la cual combina los algoritmos de Gotoh y Myers y Miller (MM) para recuperar alineamientos locales óptimos en un espacio lineal. Esta versión se ejecuta en 5 etapas, donde la primera etapa obtiene el puntaje óptimo y guarda algunas filas en el disco. Las etapas 2 a 5 ejecutan una versión modificada del MM, recuperando las coordenadas de los puntos que pertenecen al alineamiento óptimo utilizando la estrategia divide-y-vencerás. Al comparar secuencias de hasta 33 MBP en la NVIDIA GTX 285, se obtuvo un máximo de 23.63 GCUPS.

Posteriormente, en el año 2013, presentaron CUDAlign 2.1 [33], el cual calcula el alineamiento de manera similar a CUDAlign 2.0, pero con la optimización *Block Pruning* (BP), capaz de podar hasta el 53.7 % de las celdas DP. Al comparar secuencias de hasta 33 MBP en la NVIDIA GTX 560 Ti, se obtuvo un máximo de 58.21 GCUPS.

En ese mismo año, Kopar y Sikic, presentan SW# [36], implementando el algoritmo MM con la estrategia de paralelización y optimización BP para recuperar el alineamiento local entre secuencias de ADN megabase. En la GPU NVIDIA GTX 690, la comparación de secuencias de 33 MBP x 47 MBP se realizó en 23614 segundos, con una tasa de GCUPS de 65.20.

En el año 2014, Sandes y Melo presentan CUDAlign 3.0 [34] que ejecuta el algoritmo de Gotoh en múltiples GPUs con paralelismo de grano fino y entrega como resultado el puntaje óptimo. Utiliza búferes circulares para superponer cálculos y comunicaciones, conectando los nodos GPU con *sockets* TCP. Los resultados obtenidos en el clúster de GPUs Minotauro, con hasta 64 GPUs NVIDIA Tesla M2090, presentan un máximo de 1726 GCUPS al comparar secuencias de ADN de $228MBP \times 249MBP$. Esta comparación extensa tardó 9 horas y 9 minutos en completarse.

Al siguiente año, Sandes *et al.* presentan MASA [144], una arquitectura multiplataforma diseñada para ASB que emplea una técnica de poda de bloques para mejorar la eficiencia. MASA soporta variantes locales, globales y semi-globales de alineamiento en múltiples plata-

formas de hardware y software. La arquitectura de MASA se basa en CUDAlign y se destaca por su flexibilidad y personalización, lo que permite la implementación de alineadores de secuencias en diferentes plataformas con estrategias de paralelización y optimizaciones independientes de estas. Alcanza 54.74 GCUPS para secuencias con tamaños mayores a 50K usando 2 NVIDIA Tesla M2090.

En 2016, Sandes *et al.* presentan CUDAlign 4.0 [35] para el alineamiento óptimo de secuencias de ADN enormes en plataformas multi-GPU, utilizando el algoritmo SW. Esta versión introduce un nuevo algoritmo de *traceback* paralelo, denominado *Incremental Speculative Traceback*, que especula incrementalmente sobre los valores calculados hasta el momento, acelerando significativamente la fase de *traceback*. CUDAlign 4.0 logró computar matrices de hasta 60 *Peta cells* (PC), obteniendo alineamientos locales óptimos de cromosomas homólogos humanos y de chimpancés. Un ejemplo destacado es la comparación del cromosoma 5 humano y de chimpancé, que alcanzó 10370 GCUPS usando 384 GPUs, con un ratio de acierto en la especulación del 98.2 %.

En el año 2019, Zou *et al.* [145] presentan ASW (*Accelerating Smith-Waterman Algorithm*), una metodología para acelerar el algoritmo de SW en arquitecturas de CPU-GPU acopladas. Este enfoque se centra en la utilización eficiente tanto de la CPU como de la GPU en sistemas APU, optimizando la comunicación de datos y eliminando la necesidad de un bus PCI-e. ASW logra una buena tasa de rendimiento, alcanzando 7.2 GCUPS en la plataforma AMD A12. Utiliza un método de programación dinámica basado en DAG (*Directed Acyclic Graph*) para distribuir la carga de trabajo de manera eficiente entre CPU y GPU.

En el año 2020, Awan *et al.* [146] presentan ADEPT, una estrategia de alineamiento de secuencias independiente del dominio para arquitecturas GPU. Implementa el algoritmo de SW con optimizaciones específicas para GPU, no dependientes de la naturaleza de las secuencias. ADEPT es escalable en múltiples GPUs y se integra fácilmente en sistemas computacionales a gran escala. Muestra un rendimiento destacado tanto en alineamientos de proteínas como de ADN, alcanzando hasta 497 GCUPs en bases de datos de ADN y 360 GCUPs en base de datos de proteínas en un nodo con 8 GPUs, usando 8 GPUs NVIDIA V100. A pesar de su rendimiento destacado, ADEPT no representa una solución funcionalmente completa, sino que proporciona una API que puede servir como base para desarrollar una herramienta personalizada.

Referencia	Año	Pico GCUPS	Arquitecturas	Algo.	Multi-GPU	Híbrido
Weiguo Liu *et al.* [136].	2006	0.65	NVIDIA GeForce 6800 GT	Local	No	No
Yang Liu *et al.* [137].	2006	0.24	NVIDIA GeForce 7800 GTX	Local	No	No
Manavski y Valle. [138] (SW-CUDA).	2008	1.85, 3.61	NVIDIA GeForce 8800 GTX, 2×NVIDIA GeForce 8800 GTX	Local	Sí	No
Yongchao Liu *et al.* [38] (CUDASW++).	2009	9.63, 16.09	NVIDIA GeForce GTX 280, NVIDIA GeForce GTX 295	Local	Sí	No
Yongchao Liu *et al.* [39] (CUDASW++ 2.0).	2010	16.9, 29.6	NVIDIA GeForce GTX 280, NVIDIA GeForce GTX 295	Local	Sí	No
Kentie & Yongchao Liu *et al.* [134] (GASW).	2010	21.36	NVIDIA GeForce GTX 275	Local	No	No
Khalafallah *et al.* [140]	2010	12.29, 65.99	NVIDIA GeForce 9800 GT, ATI HD 5850	Local	No	No
Borovska y Lazarova [141]	2011	1.6, 7.8	NVIDIA Quadro FX3600M, NVIDIA GeForce GTX 275	Local	No	No
Yongchao Liu *et al.* [40] (CUDASW++ 3.0).	2013	119, 185.6	Intel i7 2700k 3.5Ghz + NVIDIA GeForce GTX 680, Intel i7 2700k 3.5Ghz + NVIDIA GeForce GTX 690	Local	Sí	Sí
Matija Korpar *et al.* [37] (SW#db).	2015	- [a]	Nvidia GeForce GTX 780, 2×Nvidia Tesla K80	Local	Sí	Sí
Lan *et al.* [142] (SWhybrid)	2017	220, 250	NVIDIA TITAN X, NVIDIA GTX1080	Local	Sí	Sí
Sven Warris *et al.* [143] (pyPASWAS)	2018	4.74	NVIDIA GTX 1070	Local	No	Sí
Bertil Schmidt *et al.* [41] (CUDASW++ 4.0)	2023	1940, 5010, 5710	GPU NVIDIA A100, GPU NVIDIA L40S y GPU NVIDIA H100.	Local	Sí	Sí

[a] Los autores no reportan la cantidad de GCUPS obtenidos y tampoco es posible calcularlo debido a que no están las características de la base de datos en el artículo. Sin embargo, a partir del tiempo de ejecución es posible determinar que obtuvo entre 6× a 25× mejor rendimiento que CUDASW++2.0 usando la base de datos Swiss-prot.

Tabla 2.3: Resumen de rendimiento de las implementaciones en GPU para búsqueda de similitud descritas.

Referencia	Año	Pico GCUPS	Arquitecturas	Algorit.	Multi-GPU	Híbri-do
Sandes y Melo [31] (CUDAlign 1.0).	2010	20.37	NVIDIA GTX 280	Local	Sí	No
Sandes y Melo [32] (CUDAlign 2.0).	2012	23.63	NVIDIA GTX 280	Local	Sí	No
Sandes y Melo [33] ((CUDAlign 2.1).	2013	58.21	NVIDIA GTX 560 Ti	Local	Sí	No
Kopar y Sikic [36] (SW#).	2013	65.20	NVIDIA GTX 690	Local, Global, Semi-Global, Solapa-do	Sí	No
sandes y Melo [34] (CUDAlign 3.0).	2014	1726	64 NVIDIA Tesla M2090	Local	Sí	No
Sandes *et al.* [144] (MASA).	2015	NVIDIA Tesla M2090	2XNVIDIA Tesla M2090	Local, Global y Semi-global	Sí	Sí
Sandes y Melo [35] (CUDAlign 4.0).	2016	10370	384 NVIDIA Tesla M2090	Local	Sí	Sí
Zou *et al.* [145] (AWS)	2019	7.12	AMD A12	Local	Sí	No
Awan *et al.* [146] (ADEPT).	2020	497	8 NVIDIA Tesla V100	Local	Sí	No

Tabla 2.4: Resumen de rendimiento de las implementaciones en GPU para alineamiento de a pares descritas.

Capítulo 3

Caso de Estudio y Migración a SYCL

En este capítulo se presenta un análisis exhaustivo y detallado del proceso de transición y adaptación de un software biológico escrito en CUDA al modelo de programación SYCL. Inicialmente, en la Sección 3.1, se aborda la selección del software apropiado para ASB, estableciendo el marco contextual y los criterios de elección. Posteriormente, la Sección 3.2 se enfoca en el caso de estudio seleccionado, la suite biológica SW#, proporcionando un análisis de su estructura y características. La Sección 3.3 expone las herramientas y *frameworks* relevantes que respaldan este proceso de migración, incluyendo el ecosistema de programación Intel oneAPI y herramientas de migración. En la Sección 3.4, se detalla el proceso de migración, abarcando aspectos como los errores de compilación, problemas en ejecución, y estrategias para la modernización del código. La Sección 3.5, evalúa el esfuerzo de programación mediante la métrica SLOC, y finalmente, la Sección 3.6 compara el trabajo realizado con estudios relacionados. Este capítulo, por lo tanto, ofrece una visión integral y técnica del proceso de migración de código CUDA a SYCL, destacando tanto los desafíos como las soluciones encontradas.

3.1. Selección de software para ASB

Como caso de estudio de esta investigación, se seleccionó SW#, que consiste en una suite bioinformática completa para computar ASB presentada en 2013 [36]. Esta elección se basa en varias razones:

- Frente a otras opciones (ver Sección 2.5.9), SW#:
  - Permite computar alineamiento de a pares, así como búsquedas de similitud en bases de datos, tanto para secuencias de proteínas como de ADN.
  - Permite configurar el algoritmo utilizado para diferentes tipos de alineamientos (SW, NW, Semi-global (HW) y de Solapamiento (OV)), así como las penalizaciones de apertura/extensión y la matriz de sustitución (BLOSUM45, BLOSUM50, BLOSUM62, entre otras, para proteínas; y valores de coincidencia/no coincidencia para ADN).

- Representa una implementación optimizada de ASB, al considerar técnicas de optimización como distribución dinámica de la carga de trabajo, acceso eficiente a la jerarquía de memoria de la GPU, entre otras (ver Sección 3.2).
- Combina la computación concurrente de CPU y GPU, habilitando la configuración del número de hilos de CPU y de dispositivos de GPU a utilizar.
- Se puede utilizar como una herramienta independiente o como una librería, lo que permite su integración en flujos de trabajo de bioinformática más amplios.

- Es por lo anterior, que SW# representa una herramienta de ASB que ejemplifica perfectamente su aplicación en los campos de la bioinformática y la biología computacional, los cuales han estado utilizando GPUs durante más de dos décadas.
- Además, al haber sido originalmente desarrollado en CUDA, lo convierte en un candidato ideal para estudiar la migración y evaluar diferentes aspectos de SYCL, fundamental en esta investigación. Este enfoque no solo permite evaluar la eficacia de las herramientas de migración, sino que también brinda la oportunidad de analizar la portabilidad y el rendimiento del código en diversas arquitecturas y plataformas de GPU y CPU.
- Por último, el estudio de SW# ofrece valiosas perspectivas en el campo de la bioinformática, especialmente considerando la existencia de numerosos códigos heredados basados en CUDA y la necesidad de adaptarlos a arquitecturas heterogéneas y más modernas.

3.2. Caso de estudio: SW#

El algoritmo de SW# se divide en tres fases (ver Figura 3.1):

1. Fase de resolución: utiliza la paralelización y un método de poda para calcular la puntuación máxima y el punto final del alineamiento. Emplea el método de *wavefront* basado en resolver elementos de una matriz antidiagonal al mismo tiempo, dividiendo la matriz de resolución en bloques de celdas.
2. Fase de búsqueda: encuentra el punto inicial del alineamiento resolviendo una subsecuencia inversa a partir del punto final encontrado. Se modifica el algoritmo SW original para iniciar el alineamiento en el punto final y permitir una cierta caída en las puntuaciones para asegurar que el punto de inicio esté conectado con el final.
3. Fase de reconstrucción: se centra solo en las celdas entre el inicio y el final del alineamiento, combinando el método de *wavefront* con el algoritmo modificado de Myers–Miller (MM) [147]. Esta fase se realiza en paralelo tanto en la CPU como en las GPUs.

La paralelización en SW# es un aspecto clave que permite al algoritmo realizar alineamientos de secuencias a gran velocidad, particularmente en secuencias largas, proceso que se lleva a cabo principalmente en las GPUs. A continuación, se detallan los aspectos más importantes de la paralelización en SW# [37]:

- Paralelización en GPU y CPU: SW# utiliza las capacidades de las GPUs para optimizar la fase de resolución del algoritmo de SW, permitiendo un *multithreading* masivo.

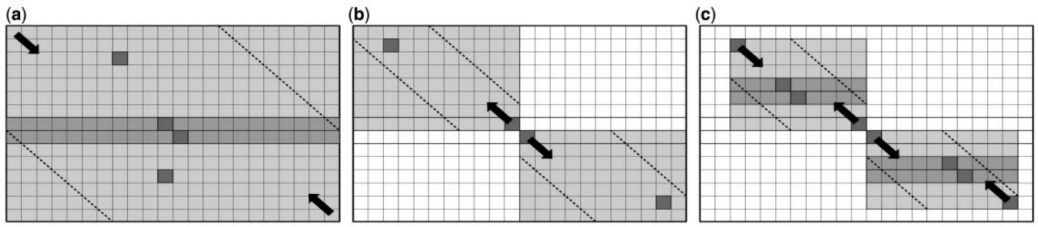

Figura 3.1: Ejemplo de SW# computando con dos GPUs. Las figuras (a-c) muestran las fases de resolución, búsqueda y reconstrucción, respectivamente. Las celdas grises representan el área ejecutable, las líneas punteadas representan el área que se puede podar y las flechas negras muestran la dirección de la ejecución. El mejor alineamiento local podría encontrarse en la parte superior, inferior o en ambas partes de la matriz. En el último caso, la puntuación total es la suma máxima de las puntuaciones de las celdas vecinas en el centro de la matriz (celdas grises más oscuras). Las posiciones de las puntuaciones máximas en cada fase están marcadas por las celdas más oscuras (extraído de [36]).

Por otro lado, en la CPU, SW# utiliza la librería OPAL [1], que permite optimizar la búsqueda de similitudes de secuencias utilizando el algoritmo de SW, mediante el uso de *multithreading* e instrucciones SIMD.

- División de secuencias en categorías cortas y largas: para administrar eficientemente la diversidad en la longitud de las secuencias de la base de datos, SW# divide las secuencias en "cortas" y "largas". En ese sentido, utiliza diferentes *kernels* para cada categoría, optimizando así la paralelización y minimizando la sobrecarga asociada con la diferencia en la longitud de las secuencias. El *kernel* corto aplica el paralelismo inter-tarea, donde procesa pares de secuencias de consulta y base de datos simultáneamente, en la que cada hilo de CUDA evalúa un par de secuencias. Por otro lado, el *kernel* largo utiliza el paralelismo intra-tarea, mediante el uso de bloques CUDA para puntuar un solo par de secuencias con múltiples hilos, aplicando un enfoque de onda antidiagonal para el alineamiento. Esta división asegura un uso eficiente de los recursos de GPU y reduce la complejidad de la memoria. La Figura 3.2 ilustra cómo cada hilo en el *kernel* largo resuelve cuatro filas.

- Optimización del uso de la memoria en la GPU: el *kernel* corto tiene una complejidad de memoria de $2 \times N \times M$, donde N es el número de secuencias cortas y M es la longitud de la secuencia más larga en esta categoría. El *kernel* largo tiene una complejidad de memoria de $9 \times N$, donde N es la suma de las longitudes de las secuencias largas. Este enfoque optimizado de uso de memoria permite un manejo más eficiente de grandes bases de datos.

- Paralelización escalable en función de la longitud de la secuencia: SW# aplica un método de alineamiento escalable donde las secuencias de la base de datos se ordenan por longitud y se procesan de manera escalonada entre la GPU y la CPU. Esto asegura que la paralelización sea efectiva independientemente de la distribución de la longitud de las secuencias en la base de datos.

[1] `https://github.com/Martinsos/opal`.

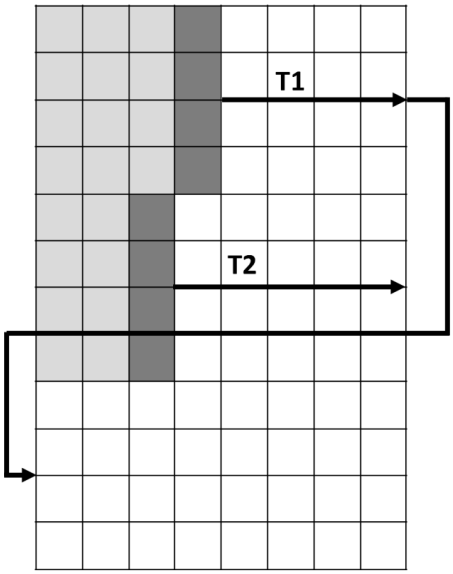

Figura 3.2: Ejemplo de *kernel* largo de SW#, donde cada hilo resuelve cuatro filas utilizando estructuras CUDA optimizadas (extraído de [37]).

- Manejo de secuencias indexadas para alineamientos selectivos: SW# permite el alineamiento de un subconjunto de secuencias de la base de datos, optimizando el proceso para minimizar la sobrecarga causada por la variabilidad en la longitud de las secuencias indexadas.
- Uso eficiente de recursos en sistemas multi-GPU y clústeres: SW# es capaz de ejecutarse en configuraciones con múltiples dispositivos GPU y en clústeres, dividiendo la base de datos y distribuyendo la carga de trabajo entre los nodos para maximizar la eficiencia.
- Soporte para múltiples consultas y tipos de alineamiento: además del algoritmo de SW, SW# soporta alineamientos globales, semi-globales y solapado, y es optimizado para manejar múltiples consultas simultáneamente, lo que lo hace adecuado para una amplia gama de aplicaciones bioinformáticas.

3.3. Herramientas y frameworks

En esta sección se listan las herramientas y frameworks utilizados en el proceso de migración.

3.3.1. Ecosistema de programación Intel oneAPI

oneAPI es una iniciativa de código abierto que propone un modelo de programación unificado para simplificar el desarrollo de aplicaciones que se ejecutan en múltiples arquitecturas de hardware, incluyendo CPUs, GPUs, FPGAs y otros aceleradores. Esta iniciativa de Intel busca proporcionar a los desarrolladores un enfoque coherente y optimizado para la

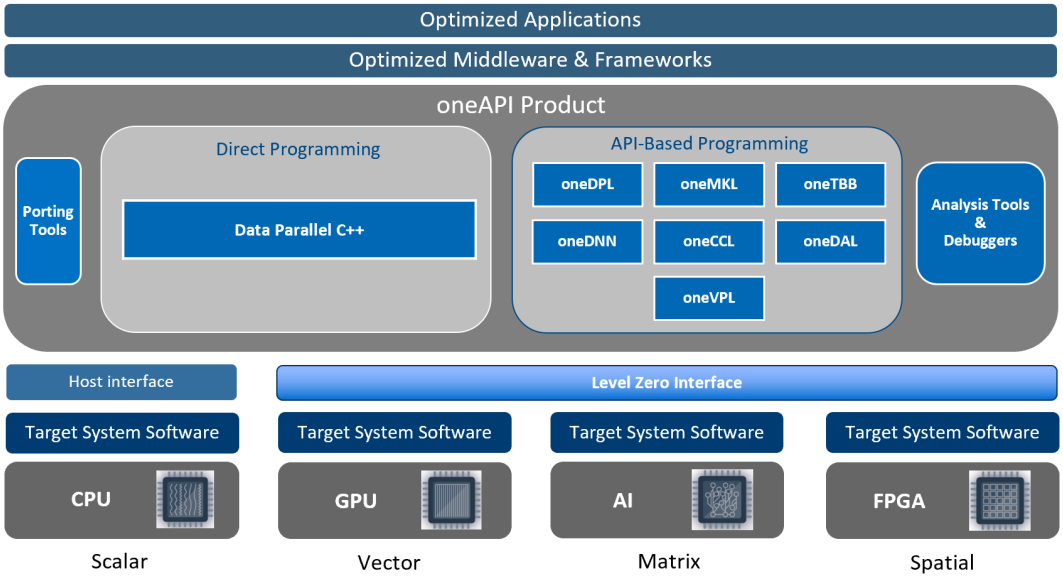

Figura 3.3: Ecosistema de oneAPI (extraído de [149]).

programación de diversos tipos de hardware, eliminando la necesidad de mantener múltiples códigos fuente y reduciendo la complejidad general del desarrollo de software [148].

La Figura 3.3 ilustra el ecosistema de oneAPI desde una perspectiva de alto nivel, resaltando la estructura y los componentes fundamentales que facilitan la programación heterogénea. Se observa una jerarquía claramente definida, comenzando con las aplicaciones y *frameworks* optimizados en la capa superior, los cuales se benefician directamente de la gama de productos ofrecidos por oneAPI.

En el núcleo del ecosistema, oneAPI se bifurca en dos principales paradigmas de programación: la programación directa y la programación basada en API. La programación directa incluye herramientas de portabilidad y DPC++. Por otro lado, la programación basada en API abarca un conjunto de bibliotecas y herramientas especializadas como oneDPL, oneMKL, oneTBB, oneDNN, oneCCL, oneDAL y oneVPL, cada una orientada a un aspecto específico de la programación paralela y el procesamiento de datos.

La interfaz de *Level Zero*, representada en la parte inferior del diagrama, actúa como una interfaz de host para el software del sistema objetivo, proporcionando una capa crítica para la interacción con el hardware. Esta capa de interfaz está directamente vinculada con el soporte para diversas arquitecturas de procesamiento, como CPU, GPU, TPU y FPGA, que representan las unidades de procesamiento escalar, vectorial, matricial y espacial, respectivamente.

El diagrama en conjunto demuestra la interoperabilidad y la naturaleza modular de oneAPI, permitiendo así que los desarrolladores utilicen una variedad de dispositivos de procesamiento bajo una interfaz de programación unificada y cohesiva. La estructura delineada es indicativa de un diseño que busca maximizar la eficiencia y la portabilidad en la programación de sistemas informáticos heterogéneos [23].

Específicamente, oneAPI se destaca por varias características clave que lo convierten en un enfoque valioso para la programación de arquitecturas heterogéneas:

- Portabilidad: oneAPI permite escribir código portable que puede ejecutarse en dife-

rentes dispositivos heterogéneos sin necesidad de realizar modificaciones significativas. Esto simplifica el proceso de desarrollo y mantenimiento de aplicaciones que aprovechan el paralelismo masivo.

- Heterogeneidad: oneAPI aborda la heterogeneidad de los sistemas al proporcionar modelos de programación que permiten aprovechar las capacidades específicas de cada dispositivo sin requerir conocimientos profundos de su arquitectura.
- Rendimiento: oneAPI ofrece diferentes librerías de rendimiento optimizado y controladores específicos del dispositivo (como oneCCL, oneDAL, oneDNN, oneMKL, oneTBB, and oneVPL, entre otros), lo que permite aprovechar al máximo el potencial de cómputo de cada dispositivo y aumentar el rendimiento de las aplicaciones.
- Ecosistema de herramientas y librerías: oneAPI cuenta con un amplio conjunto de herramientas y librerías que facilitan el desarrollo de aplicaciones heterogéneas. Estas herramientas incluyen compiladores, depuradores, analizadores de rendimiento y librerías específicas para tareas como procesamiento de imágenes, aprendizaje automático, simulación, entre otras.

3.3.2. Herramienta de migración

La herramienta de migración utilizada en esta tesis es SYCLomatic, anteriormente conocida como DPCT (*Data Parallel C++ Compiler Tool*). SYCLomatic es una herramienta desarrollada por Intel, incluida como una herramienta de compatibilidad dentro del ecosistema de oneAPI, que está diseñada para facilitar la migración de código CUDA a SYCL.

SYCLomatic proporciona un enfoque automatizado para transformar el código CUDA existente en código SYCL, lo que permite a los desarrolladores aprovechar las ventajas del estándar SYCL sin tener que reescribir completamente sus aplicaciones desde cero.

La migración de código CUDA a SYCL es especialmente relevante debido a la gran cantidad de código CUDA heredado que existe actualmente, junto con la creciente popularidad de SYCL como un enfoque de programación unificado para sistemas heterogéneos. Sin una herramienta de migración automática, el esfuerzo de migrar aplicaciones CUDA extensas a SYCL puede ser muy grande.

En este sentido, SYCLomatic simplifica el proceso de migración al automatizar gran parte de la tarea. La herramienta realiza análisis estático del código CUDA existente y aplica las transformaciones necesarias para convertirlo en código SYCL equivalente, lo que incluye la identificación y traducción de llamadas de API específicas de CUDA, como funciones de memoria y sincronización, a las equivalencias correspondientes en SYCL.

Es importante tener en cuenta que SYCLomatic es una herramienta que asiste en la migración de código CUDA a SYCL, pero no garantiza una conversión perfecta en todos los casos. En particular, suele migrar entre un 80 % a un 90 % del código, dependiendo de la complejidad del mismo, por lo cual es posible que aún sea necesario realizar ajustes manuales en el código resultante para asegurar un comportamiento correcto y un rendimiento óptimo en el nuevo entorno SYCL [150].

3.4. Proceso de migración

El proceso de migración se puede dividir en 6 etapas, siendo opcionales las 2 últimas:

1. Ejecutar la herramienta SYCLomatic para generar la primera versión del código.

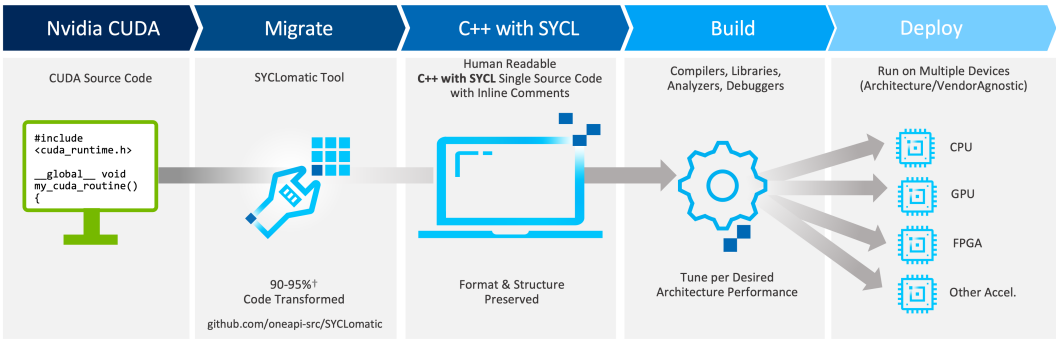

Figura 3.4: Proceso de migración desde código CUDA a su equivalente código SYCL ejecutado en los aceleradores finales (extraído de [24]).

2. Modificar el código migrado en base a los errores y las advertencias reportadas por SYCLomatic para obtener la primera versión ejecutable.
3. Corregir los errores de tiempo de ejecución para obtener la primera versión funcional.
4. Verificar la corrección de los resultados y corregir de ser necesario.
5. Opcional: estandarizar el código resultante, considerando que algunas de las instrucciones generadas por SYCLomatic dependen del ecosistema oneAPI.
6. Opcional: Optimizar el código resultante, considerando optimizaciones específicas de SYCL o de la aplicación.

La Figura 3.4 ilustra de forma gráfica el flujo que comienza con un código CUDA original, continúa con su traducción a SYCL y finaliza con la ejecución en los aceleradores finales.

3.4.1. Errores de compilación y alertas

Tras obtener la primera versión migrada de SW#, se reportaron las siguientes advertencias informadas por SYCLomatic:

```
1 DPCT1003: Migrated API does not return error code. (*, 0) is inserted. You may
      need to rewrite this code
2
3 DPCT1009: SYCL uses exceptions to report errors and does not use the error
      codes. The original code was commented out and a warning string was
      inserted. You need to rewrite this code.
```

Ambas advertencias ocurren al utilizar funciones nativas de CUDA, como por ejemplo, los códigos de error internos del lenguaje (Figura 3.5a), técnica que por lo general se utiliza al intercambiar datos con el dispositivo. Dado que SYCLomatic no puede traducirlas, modifica el código para mantenerlo funcional (Figura 3.5b). Las Figuras 3.5a y 3.5b muestran asignaciones de memoria en la GPU usando CUDA y SYCL, respectivamente. Por defecto, SYCLomatic intenta utilizar el modelo USM debido a que produce un código más compacto y permite que SYCLomatic sea compatible con un mayor número de APIs relacionadas con la memoria.

```
1 DPCT1005: The SYCL device version is different from CUDA Compute Compatibility
      . You may need to rewrite this code.
```

```
size_t valuesSize =
    databaseLen * sizeof(double);
double* valuesGpu;

CUDA_SAFE_CALL(
    cudaMalloc(
        &valuesGpu, valuesSize
    )
);
```

(a) CUDA

```
size_t valuesSize =
    databaseLen * sizeof(double);
double* valuesGpu;

CUDA_SAFE_CALL((
 valuesGpu = (double *)sycl::malloc_device(
    valuesSize, dpct::get_default_queue()),
0));
```

(b) SYCL

Figura 3.5: Ejemplo de `CUDA_SAFE_CALL`.

Este problema está relacionado con el anterior y surge al consultar atributos intrínsecos de CUDA. Aunque SYCLomatic puede obtener información de la GPU, como el número de registros o el tamaño máximo de memoria, entre otros [2], algunos atributos propios de CUDA (por ejemplo, información del controlador de CUDA) no son traducibles. La Figura 3.6a muestra que, en el código original, el número de bloques e hilos de CUDA depende de la versión del controlador. La Figura 3.6b presenta el código migrado, mostrando que es posible obtener información sobre las propiedades de la GPU, con la excepción de aquellas específicas de CUDA.

```
cudaDeviceProp properties;
cudaGetDeviceProperties(
    &properties, card
);

bool major = properties.major < 2;
int threads = major ? 64 : 128;
int blocks = major ? 360 : 480;
```

(a) CUDA

```
dpct::device_info properties;

dpct::dev_mgr::instance()
    .get_device(card)
    .get_device_info(properties);

bool major = false;
int threads = major ? 64 : 128;
int blocks = major ? 360 : 480;
```

(b) SYCL

Figura 3.6: Consultas de propiedades del dispositivo.

```
DPCT1049: The workgroup size passed to the SYCL kernel may exceed the limit.
    To get the device limit, query info::device::max_work_group_size. Adjust
    the workgroup size if needed.
```

Para ejecutar el *kernel* de CUDA, es necesario configurar tanto los tamaños de bloque como de hilo; sin embargo, cada dispositivo tiene un límite de tamaño diferente. SYCLomatic alerta al programador que el código migrado puede exceder el límite máximo de WG que soporta la arquitectura subyacente. Además, recomienda ajustar el código si es necesario. La Figura 3.7a ilustra cómo ejecutar el *kernel* en CUDA, mientras que la Figura 3.7b muestra el equivalente en SYCL.

```
DPCT1065: Consider replacing sycl::nd_item::barrier() with sycl::nd_item::
    barrier(sycl::access::fence_space::local_space) for better performance if
    there is no access to global memory.
```

En esta situación, SYCLomatic recomienda al programador utilizar un parámetro adicional al sincronizar los hilos dentro del *kernel*, siempre y cuando no se utilice memoria global.

[2] `https://docs.oneapi.io/versions/latest/dpcpp/iface/device.html`.

```
1   solveShort<<<blocks, threads>>>(...);
```

(a) CUDA

```
1   dpct::get_default_queue()
2       .submit([&](sycl::handler &cgh) {
3    ...
4   cgh.parallel_for(
5     sycl::nd_range<3>
6       (sycl::range<3>(1, 1, blocks) *
7       sycl::range<3>(1, 1, threads),
8       sycl::range<3>(1, 1, threads)),
9     [=](sycl::nd_item<3> item_ct1) {
10      solveShort(...);
11    });
12  });
```

(b) SYCL

Figura 3.7: Lanzamiento de *kernel* con tamaño de WG dinámico.

Por defecto, la herramienta no optimiza automáticamente este aspecto porque no puede discernir si se está utilizando esta memoria. Un ejemplo de la sincronización de hilos en CUDA y el código migrado a SYCL puede verse en las Figuras 3.8a y 3.8b, respectivamente.

```
1   ...
2   __syncthreads();
3   ...
```

(a) CUDA

```
1   ...
2   item_ct1.barrier();
3   ...
```

(b) SYCL

Figura 3.8: Sincronización entre hilos.

```
DPCT1084: The function call has multiple migration results in different
    template instantiations that could not be unified. You may need to adjust
    the code.
```

En CUDA, las funciones genéricas son una forma común de reducir el tamaño del código, ya que permiten la reutilización del mismo para datos de diferentes tipos. Aunque SYCL soporta esta característica de programación, no puede migrar automáticamente este tipo de código debido a la multiplicidad de opciones de migración posibles. La Figura 3.9a muestra un ejemplo de CUDA en el que las instrucciones dependen del tipo de parámetro enviado a la función del *kernel*. La Figura 3.9b muestra el código migrado correspondiente.

```
DPCT1059: SYCL only supports 4-channel image format. Adjust the code.
```

En CUDA, las variables de memoria de textura pueden asignarse a través de 1 a 4 canales, mientras que en SYCL, la memoria de textura se accede a través de imágenes. Como reporta la advertencia de SYCLomatic, SYCL solo soporta el uso de imágenes de 4 canales, por lo que el programador debe adaptar las partes del código en las que se utilizan imágenes de diferentes tamaños. En la Figura 3.10a, se declara una variable de textura de 1 canal en CUDA (ubicada en el dispositivo) y finalmente se lee un dato de ella. La Figura 3.10b presenta un posible ajuste al código correspondiente para convertir una variable de textura de 1 canal en su equivalente de 4 canales. Como se puede observar, esta transformación implica la alteración de los índices mediante los cuales se accede a la memoria con el fin de obtener los datos correctos. Por lo tanto, requiere efectuar un desplazamiento a la derecha de 2 bits (equivalente a `DIV 4`) combinado con una operación lógica `AND 3` (equivalente a `MOD 4`) en la operación de lectura correspondiente.

```cuda
class SubVector {
public:
  __device__ int operator()(...) {
    ...
  }
};

template <class Sub>
__global__ static void solveLong(
..., Sub sub) {
    sub(...);
}

solveLong<<<blocks, threads>>>(
 ..., SubVector());
```

(a) CUDA

```sycl
class SubVector {
public:
  int operator()(...) {
      ...
  }
};

template <class Sub>
static void solveLong(..., Sub sub) {
    sub(...);
}

cgh.parallel_for(
  sycl::nd_range<3>
    (sycl::range<3>(1, 1, blocks) *
    sycl::range<3>(1, 1, threads),
    sycl::range<3>(1, 1, threads)),
  [=](sycl::nd_item<3> item_ct1) {
    solveLong(..., SubVector());
  });
});
```

(b) SYCL

Figura 3.9: Funciones genéricas.

```cuda
texture<char> colTexture;

int colSize = colsGpu * sizeof(char);
char *colGpu;
cudaMalloc(&colGpu, colSize);
cudaMemcpy(colGpu, colCpu,
    colSize, TO_GPU);
cudaBindTexture(NULL, colTexture,
    colGpu, colSize);

char v = tex1Dfetch(colTexture, 10);
```

(a) CUDA

```sycl
//dpct::image_wrapper<char, 1> colTexture;
dpct::image_wrapper<sycl::char4, 1> colTexture;
int colSize = colsGpu * sizeof(char);
char* colGpu;

colGpu = (char *)sycl::malloc_device(
    colSize, dpct::get_default_queue());

dpct::get_default_queue()
    .memcpy(colGpu, colCpu, colSize).wait();

colTexture.attach(colGpu, colSize);

// DIV 4 y MOD 3
char v = colTexture.read(10 >> 2)[10 & 3];
```

(b) SYCL

Figura 3.10: Memoria de textura de 4 canales.

3.4.2. Errores en ejecución

Una vez que el código migrado compila sin errores, es necesario realizar una verificación para asegurarse de que no haya errores de ejecución y de que los resultados obtenidos sean correctos (es decir, ante un mismo contexto, ambos códigos deben generar los mismos resultados). En esta situación particular, a pesar de que el código SYCL compiló correctamente, se produjo el siguiente error durante la ejecución:

```
For a 1D/2D image/image array, the width must be a Value >= 1 and <=
    CL_DEVICE_IMAGE2D_MAX_WIDTH.
```

Este error surge porque las imágenes SYCL tienen un tamaño limitado, siendo el tamaño máximo de las 1D (vectores) menor que las 2D (matrices). Para resolver este problema, el objeto de imagen debe convertirse en otra abstracción de memoria de SYCL, ya sea *buffers* o USM. En este punto se optó por utilizar USM para continuar con la ideología de SYCLomatic en cuanto al uso del modelo unificado.

La Figura 3.11a muestra cómo se asigna una memoria de textura de 2 niveles en la GPU, mientras que la Figura 3.11b ilustra cómo usar USM para enviar un vector al dispositivo. De esta manera, el mecanismo de lectura también cambia, tanto en CUDA (Figura 3.12a) como en SYCL (Figura 3.12b).

```
texture<int, 2,
  cudaReadModeElementType> seqsTexture;

cudaArray *sequencesGpu;
cudaChannelFormatDesc channel =
  seqsTexture.channelDesc;
cudaMallocArray(&sequencesGpu,
    &channel, sequencesCols, sequencesRows);
cudaMemcpyToArray(sequencesGpu, 0, 0,
    sequences, sequencesSize, TO_GPU);
cudaBindTextureToArray(
    seqsTexture, sequencesGpu);
```

(a) CUDA

```
static int *seqsGpu;

seqsGpu = (int *)sycl::malloc_device(
 sequencesCols * sequencesRows * sizeof(int),
 dpct::get_default_queue());

dpct::get_default_queue()
 .memcpy(seqsGpu, sequences, sequencesCols *
        sequencesRows * sizeof(int))
  .wait();
```

(b) SYCL

Figura 3.11: Memoria de textura 2D CUDA, adaptado usando USM SYCL.

```
int columnCodes = tex2D(
  seqsTexture, colOff, j + rowOff);
```

(a) CUDA

```
int columnCodes =
    seqsGpu[(j + rowOff) * sequencesCols + colOff];
```

(b) SYCL

Figura 3.12: Acceso de datos en un arreglo 2D.

Finalmente, la Figura 3.13 resume las advertencias generadas por SYCLomatic, agrupadas en 4 áreas: manejo de errores (DPCT1003), funcionalidades no soportadas (DPCT1005, DPCT1084 y DPCT1059), recomendaciones (DPCT1049) y optimizaciones (DPCT1065):

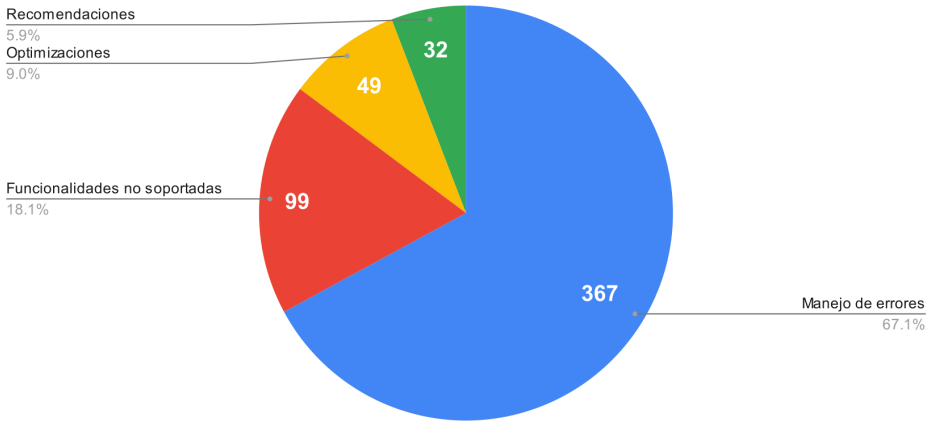

Figura 3.13: Distribución de las advertencias generadas por SYCLomatic.

3.4.3. Verificación funcional

Tras finalizar el proceso de migración, se realizaron diferentes pruebas, tanto para secuencias de proteínas como de ADN, utilizando distintos algoritmos de alineamiento y esquemas de puntuación (presentados en la Sección 4 de resultados). Finalmente, se verificó que tanto CUDA como SYCL produjeron los mismos resultados.

3.4.4. Modernización de código y optimizaciones

El código SW# fue desarrollado exclusivamente para GPUs de NVIDIA y está particularmente diseñado para aquellas lanzadas a mediados de 2010. Algunas configuraciones están indicadas estáticamente en el código, como por ejemplo, las dimensiones de los bloques para los *kernels*. Esto conlleva dos limitaciones al ejecutar el código migrado en otros dispositivos: primero, el código no aprovecha completamente las GPUs actuales de NVIDIA, que presentan mayor capacidad de memoria y potencia de cálculo. Segundo, impide la ejecución en dispositivos con diferentes requisitos de WG, como GPUs de otros fabricantes o distintas arquitecturas como CPUs.

Para remediar este problema, la configuración estática del tamaño del WG fue reemplazada por una configuración dinámica que considera las longitudes de secuencia y el valor máximo permitido por el dispositivo correspondiente [3]. De esta manera, el soporte del código migrado se extiende a dispositivos de diferentes arquitecturas.

3.4.5. Estandarización a SYCL (opcional)

Aunque SYCLomatic produce código DPC++, que implementa la interfaz SYCL, utiliza instrucciones que dependen del ecosistema oneAPI. Por ejemplo, en la Figura 3.7b, el código migrado consulta atributos del dispositivo utilizando funciones específicas de DPC++. Es por esto que se deben realizar algunos ajustes manuales para lograr un código SYCL completamente portable. Por un lado, las variables de memoria constante fueron reemplazadas por argumentos del *kernel*, los cuales aún residen en la memoria constante al ejecutarse en GPUs [4]. Por otro lado, las funciones específicas de DPC++ fueron reemplazadas por llamadas puras de SYCL para consultar la información del dispositivo. Como resultado, esta versión final del código puede compilarse con cualquiera de las implementaciones SYCL [5].

3.5. Evaluación del esfuerzo de programación

El esfuerzo de programación en este proceso de migración de código está directamente relacionado a la eficiencia de SYCLomatic, que en este contexto se refiere a qué tan buena es la herramienta para traducir automáticamente el código de CUDA a SYCL. En particular, esta cuestión se evalúa midiendo las SLOC de las versiones de CUDA y SYCL [6] (véase la

[3] Es importante señalar que la misma mejora también se aplicó al código original de CUDA para evitar sesgos en la evaluación del rendimiento.

[4] Es importante notar que este cambio implicó una reducción significativa en el número de líneas de código.

[5] Afortunadamente, varios están disponibles de un número creciente de proveedores `https://www.khronos.org/sycl/`.

[6] Para medir SLOC, se utilizó la herramienta `cloc` (disponible en `https://github.com/AlDanial/cloc`.), excluyendo líneas en blanco y comentarios.

Tabla 3.1). La versión original de SW# presenta 8072 SLOC. Después de ejecutar SYCLomatic, se encontró que 407 SLOC de CUDA no fueron migradas automáticamente. Para alcanzar la primera versión funcional, fueron necesarias algunas modificaciones manuales, aumentando las SLOC a 12175. En resumen, SYCLomatic logró migrar el 95 % del código de CUDA, confirmando las afirmaciones de Intel. Sin embargo, fue necesario añadir 1718 SLOC (+21 %) al resultado de SYCLomatic para obtener la primera versión ejecutable. Finalmente, al eliminar parte del código específico de SYCLomatic (estandarización de SYCL), las SLOC de SYCL se redujeron aproximadamente en un 20 %.

Tabla 3.1: SLOC de SW# (versiones CUDA y SYCL).

CUDA[1]		SYCL[2]		
Original	No migrado	Resultado SYCLomatic	Manuales	SYCL puro
8072	408	10457	12175	9866

El análisis del proceso de migración con SYCLomatic muestra una eficiencia destacada en la traducción automática del código de CUDA a SYCL, logrando migrar un 95 % del código, confirmando lo promocionado por Intel en cuanto a la efectividad de la herramienta. No obstante, esta eficiencia se ve atenuada por la necesidad de intervenciones manuales significativas, como se evidencia en el aumento del 21 % en las SLOC para obtener una versión funcional. A pesar de este requisito adicional, la herramienta demuestra ser una solución valiosa en el ámbito de la migración de código, ya que facilita en gran medida la transición entre plataformas de cómputo.

3.6. Trabajos relacionados y discusión

Algunos estudios preliminares que evalúan la portabilidad de SYCL y oneAPI se pueden encontrar en simulación [151], matemáticas [152], aprendizaje automático [153, 154], pruebas de software [155, 156], procesamiento de imágenes [157] y criptografía [158]. En el campo de la bioinformática, también se pueden mencionar algunos trabajos: en [16], los autores describen la experiencia de traducir una implementación CUDA de un algoritmo de detección de epistasis de alto orden a SYCL, encontrando que el rendimiento más alto de ambas versiones es comparable en una GPU NVIDIA V100. Es importante destacar que se requirieron de optimizaciones particulares en los *kernels* de la implementación SYCL para alcanzar su rendimiento máximo.

En [159], los autores migran *kernels* representativos en aplicaciones de bioinformática de CUDA a SYCL y evalúan su rendimiento en una GPU NVIDIA V100, explicando las diferencias de rendimiento a través de *profiling* y análisis de código. En general, CUDA obtiene mejores rendimientos y los autores lo relacionan con su entorno de desarrollo maduro y extenso. Al igual que en el trabajo anterior, los autores no informaron si se empleó migración manual o automática.

En [160], los autores evalúan el rendimiento y la portabilidad del *kernel* ADEPT basado en CUDA para el alineamiento de secuencias cortas (*short-reads*) de SW. A diferencia de este estudio, los autores realizaron una migración manual para obtener una versión equivalente de ADEPT en DPC++, argumentando que el código resultante era innecesariamente complejo y requería cambios importantes.

En [161], los autores traducen el software de acoplamiento molecular AutoDock-GPU de CUDA a SYCL utilizando la versión previa de SYCLomatic (*dpct*), destacando, al igual

que ocurre en el estudio actual, que esta herramienta reduce en gran medida el esfuerzo de migración de código, pero aún se requieren modificaciones manuales para completar y ajustar el mismo.

En [162], los autores presentaron *OneJoin*, una herramienta basada en oneAPI para editar la unión de similitud en la decodificación de datos de ADN. Esta herramienta fue desarrollada desde cero utilizando oneAPI y se comprobó su portabilidad en diferentes plataformas.

En [163], los autores migran una aplicación de modelado sísmico basada en CUDA a SYCL utilizando SYCLomatic y oneAPI de Intel. El estudio muestra que, aunque SYCLomatic facilitó la migración automatizando la mayoría de las tareas, fueron necesarios ajustes manuales para asegurar la corrección funcional del código. La optimización posterior llevó a un rendimiento comparable al código CUDA en GPUs de NVIDIA y superior en GPUs de Intel, destacando la utilidad de SYCL para desarrollar código fuente único eficiente en arquitecturas heterogéneas.

En [164], el autor detalla el proceso de migración manual de un algoritmo de coloreo de grafos paralelos de CUDA a SYCL. Durante este proceso, se llevó a cabo la tarea de mapear las funciones de CUDA a SYCL y se evaluó el rendimiento de los *kernels* en las GPUs NVIDIA P100 y V100. Los resultados obtenidos demostraron que ambos modelos presentan un rendimiento comparable. Sin embargo, se observó que algunas características específicas de CUDA aún no son compatibles con SYCL, como ciertas funciones de propiedades del dispositivo, matemáticas y primitivas de *warps*; la función `__launch_bounds__()` para especificar el número máximo de hilos por bloque y la cantidad requerida de bloques por multiprocesador; la configuración preferida de la caché, como la configurada con `cudaFuncSetCacheConfig()` en CUDA, para dispositivos que comparten la caché L1 y la memoria local compartida; funciones de depuración como el "`printf()`" dentro del *kernel*; y características específicas de la arquitectura de CUDA. A pesar de esto, la migración permitió evaluar el rendimiento en diferentes plataformas, incluyendo una GPU Intel con interfaces OpenCL y Level Zero. En conclusión, los autores consideran que SYCL es un modelo prometedor para la computación heterogénea, ya que combina la portabilidad y eficiencia de OpenCL con la flexibilidad del C++ de fuente única. No obstante, es necesario tener un buen entendimiento de ambos modelos para llevar a cabo una migración efectiva.

En el presente estudio, se logró migrar una suite biológica completa basada en CUDA a SYCL utilizando la herramienta de compatibilidad SYCLomatic de Intel. Aunque la herramienta tradujo la mayor parte del código CUDA, fue necesario realizar intervenciones manuales para completar el proceso y obtener un código SYCL compatible. A diferencia de los trabajos anteriores, se llevó a cabo una estandarización del código migrado a SYCL puro, con el objetivo de no depender de las librerías de oneAPI y así poder utilizar otras implementaciones SYCL. Para evaluar de alguna manera cuantitativa el esfuerzo de programación, se analizó la métrica SLOC entre el código CUDA original y el código SYCL migrado.

Como conclusión, la herramienta SYCLomatic se presenta como una solución de gran utilidad. Su principal ventaja radica en la capacidad para migrar una gran parte del código que sigue patrones comunes en la conversión de CUDA a SYCL. Esto es fundamental, ya que reduce los errores humanos y disminuye significativamente el tiempo y el esfuerzo requeridos en la programación. Sin embargo, es importante tener en cuenta que SYCLomatic no es una solución completa por sí sola. En ciertas situaciones donde no hay un equivalente directo para ciertas funciones o segmentos de código de CUDA en SYCL, se necesitará de intervención del programador. Esto implica que en varios casos se deben realizar modificaciones y optimizaciones manuales para adaptar adecuadamente el código.

Por otro lado, los estudios relacionados revelan que, aunque SYCLomatic es una he-

rramienta útil, no es indispensable para la migración de código CUDA a SYCL. Como fue observado en investigaciones anteriores, los desarrolladores pueden optar por realizar una migración manual, lo que les permite explorar diferentes estrategias de optimización, como el uso de *buffers* en lugar de USM. Esta aproximación completamente manual puede ser especialmente ventajosa en términos de optimización específica del código para la aplicación objetivo. Adicionalmente, también puede conducir a un código resultante más sintético y *elegante*, considerando que no será producto de una traducción automática. Sin embargo, se debe tener en cuenta que el esfuerzo de programación será significativamente mayor, especialmente en aplicaciones de gran cantidad de líneas de código. En estos escenarios, podría ser más productivo modificar el código ya migrado por SYCLomatic en lugar de comenzar el proceso de migración desde cero.

Capítulo 4

Resultados Experimentales

En este capítulo se expone el análisis de los resultados obtenidos en la presente investigación. Comienza con la Sección 4.1, donde se detalla la metodología aplicada y las configuraciones para llevar a cabo los experimentos. Posteriormente, la Sección 4.2 desglosa en profundidad los resultados obtenidos, abordando aspectos clave como el rendimiento, la funcionalidad, y la portabilidad en diferentes plataformas y configuraciones de hardware, incluyendo distintas GPUs y CPUs. Esta sección proporciona una visión integral del comportamiento del código SW# SYCL en diferentes dispositivos. Finalmente, la Sección 4.3, realiza una comparativa con trabajos previos y discute las implicaciones de los resultados obtenidos.

4.1. Diseño experimental

A continuación se detallan varias cuestiones vinculadas al diseño experimental, incluyendo las plataformas utilizadas, las configuraciones tanto del SW# como de los compiladores utilizados, el detalle de las pruebas realizadas y los objetivos del análisis.

4.1.1. Hardware

Se llevaron a cabo pruebas en un conjunto amplio y diverso de plataformas equipadas con GPUs. En particular, se consideraron 8 dGPUs (6 de NVIDIA, 1 de AMD, 1 de Intel) y 3 iGPUs (2 de Intel, 1 de AMD). Además, se incluyeron 7 CPUs de Intel (de diferentes segmentos) y 1 CPU de AMD. Los detalles específicos de estas plataformas se pueden encontrar en la Tabla 4.1.

4.1.2. Software

Para poder ejecutar el código SYCL en GPUs de NVIDIA y AMD, se requirieron algunas modificaciones en el proceso de compilación para oneAPI, ya que al día en el que se hicieron los experimentos, no se admitían por defecto estas plataformas [1]. Sin embargo, recientemente Codeplay presentó *plugins* binarios gratuitos[2] para dar soporte a estas GPUs. Tras

[1] `https://intel.github.io/llvm-docs/GetStartedGuide.html`

[2] `https://codeplay.com/portal/blogs/2022/12/16/bringing-nvidia-and-amd-support-to-oneapi.html`

las modificaciones, fue posible ejecutar código DPC++ en GPUs de NVIDIA y AMD utilizando el compilador Clang++ (`16.0`). Es importante aclarar que se intentó incluir GPUs NVIDIA más antiguas (por ejemplo, una basada en Kepler), dado que SW# es un software diseñado para versiones CUDA antiguas, pero no fue posible debido a que oneAPI solo admite GPUs NVIDIA desde la línea Maxwell en adelante, impidiendo su inclusión en la comparación de rendimiento. Las versiones utilizadas de oneAPI y CUDA fueron las 2023.0.0 y 11.7, respectivamente. Para la comparación entre diferentes implementaciones SYCL, se utilizó AdaptiveCPP como alternativa, en su versión 23.10.0 construida a partir del repositorio público [3] con clang-v15.0, CUDA v11.7 y ROCm v5.4.3. La selección de oneAPI y AdaptiveCPP se debe a que ambos representan las versiones más sólidas y completas actualmente disponibles. Por último, SW# fue compilado tanto en CUDA, como en oneAPI y AdaptiveCpp usando el nivel de optimización *O3*.

4.1.3. Pruebas

Para los alineamientos de proteínas, se utilizaron las siguientes bases de datos y configuraciones:

- Base de datos Swiss-Prot (versión 2022_07)[4]: la base de datos contiene 204173280 residuos de aminoácidos en 565928 secuencias con una longitud máxima de 35213.
- Base de datos Env. NR (versión 2021_04)[5]: la base de datos contiene 995210546 residuos de aminoácidos en 4789355 secuencias con una longitud máxima de 16925.
- Las secuencias de consulta varían en longitud de 144 a 5478, y fueron extraídas de la base de datos Swiss-Prot (números de acceso: *P02232, P05013, P14942, P07327, P01008, P03435, P42357, P21177, Q38941, P27895, P07756, P04775, P19096, P28167, P0C6B8, P20930, P08519, Q7TMA5, P33450 y Q9UKN1*).
- La matriz de sustitución seleccionada es BLOSUM62 y los puntajes de inserción y extensión de *gaps* se establecieron en 10 y 2, respectivamente.

Para los alineamientos de ADN, la Tabla 4.2 presenta los números de acceso y tamaños de las secuencias utilizadas. Los parámetros de puntuación utilizados fueron +1 para coincidencias, -3 para no coincidencias, -5 para *gap* de apertura y -2 para *gap* de extensión.

Para eliminar el impacto de la CPU en el rendimiento, SW# se configuró en modo solo GPU (con el *flag* `T=0`). Por otro lado, se configuraron diferentes tamaños de WG para obtener el óptimo. Finalmente, cada prueba se ejecutó 20 veces, y el rendimiento se calculó como el promedio para minimizar la variabilidad.

4.1.4. Objetivos

Los experimentos desarrollados en la siguiente sección buscan evaluar la portabilidad funcional y de rendimiento de SYCL para el caso de estudio de la suite de SW#. Con este objetivo, primeramente en la Sección 4.2.1 se evalúa la efectividad del código migrado en las diferentes variantes de ejecución que ofrece SW#, comparando el rendimiento entre CUDA y SYCL en las GPUs NVIDIA. Luego, en la Sección 4.2.2, se explica el modelo de rendimiento aplicado, para posteriormente hacer un análisis de portabilidad funcional y de

[3]Proyecto AdaptiveCpp: `https://github.com/AdaptiveCpp/AdaptiveCpp`
[4]Swiss-Prot: `https://www.uniprot.org/downloads`
[5]ENV NR: `https://ftp.ncbi.nlm.nih.gov/blast/db/`

Tabla 4.1: Plataformas utilizadas en los experimentos.

CPU			GPU			
ID	Procesador (Segmento)	RAM (Mem.)	ID	Fabricante (Tipo)	Modelo (Arq.)	GFLOPS Pico (SP)
Xeon E5-2	Intel Xeon E5-2695 V3 (servidor)	64 GB	GTX 980	NVIDIA (Discreta)	GTX 980 (Maxwell)	5000
			GTX 1080	NVIDIA (Discreta)	GTX 1080 (Pascal)	8873
Xeon Gold	Intel Xeon Gold 6138 (servidor)	64 GB	V100	NVIDIA (Discreta)	V100 (Volta)	14130
		64 GB	RTX 3090	NVIDIA (Discreta)	RTX 3090 (Ampere)	35580
Core i5-7	Intel Core i5-7400 (escritorio)	8 GB	RTX 2070	NVIDIA (Discreta)	RTX 2070 (Turing)	7465
Core i5-10	Intel Core i5-10400F (escritorio)	12 GB	RTX 3070	NVIDIA (Discreta)	RTX 3070 (Ampere)	20310
Core i9-9	Intel Core i9-9900K (escritorio)	65 GB	P630	Intel (Integrada)	UHD Graphics P630 (Gen 9.5)	441.6
Core i9-13	Intel Core i9-13900k (escritorio)	65 GB	ARC770	Intel (Discreta)	A770 (Xe HPG)	19660
			UHD770	Intel (Integrada)	UHD Graphics 770 (Gen12.2)	35.2
Xeon E5-1	Intel Xeon E5-1620 V3	32 GB	RX6700	AMD (Discreta)	RX 6700 XT (RDNA2)	13215
Ryzen3	AMD Ryzen 3 5300U (mobile)	12 GB	RX Vega 6	AMD (Integrada)	RX Vega 6 (Vega)	845.6

Tabla 4.2: Información de secuencias de ADN usados en los experimentos

Secuencia 1[1]		Secuencia 2[2]		Tamaño de matriz (celdas)
Acceso	Tamaño	Acceso	Tamaño	
CP000051.1	1M	AE002160.2	1M	1G
BA000035.2	3M	BX927147.1	3M	9G
AE016879.1	5M	AE017225.1	5M	25G
NC_005027.1	7M	NC_003997.3	5M	35G
NC_017186.1	10M	NC_014318.1	10M	100G

rendimiento a través de diferentes fabricantes de GPUs, y también distintas arquitecturas como CPUs: en la Sección 4.2.3, se realiza un análisis de portabilidad utilizando GPUs de NVIDIA, Intel (iGPU y dGPU) y AMD (iGPU y dGPU). En la Sección 4.2.4 se extiende la evaluación previa mediante la ejecución multi-GPU. En la Sección 4.2.5 se hace una evaluación de portabilidad funcional y de rendimiento en arquitecturas CPUs de Intel y AMD. Posteriormente, en la Sección 4.2.6 se combinan las CPUs y GPUs para hacer un análisis de portabilidad en un contexto híbrido. Finalmente, en la Sección 4.2.7 se realiza una comparación entre 2 diferentes implementaciones SYCL, seleccionando una plataforma de cada tipo.

4.2. Resultados de rendimiento y portabilidad

En esta sección se presenta un análisis exhaustivo de los resultados obtenidos en términos de rendimiento y portabilidad. En la Sección 4.2.1 se realiza un análisis detallado del rendimiento y la portabilidad funcional del SW#. A continuación, en la Sección 4.2.2 se expone el modelo utilizado para evaluar la portabilidad del rendimiento en los análisis siguientes. La Sección 4.2.3 se dedica a examinar la portabilidad de las GPUs de forma individual. Por su parte, la Sección 4.2.4 aborda la portabilidad en entornos multi-GPU. Posteriormente, en la Sección 4.2.5 se evalúa la portabilidad de rendimiento en CPUs. A continuación, en la Sección 4.2.6 se estudia la portabilidad de rendimiento en entornos híbridos que combinan CPUs y GPUs. Por último, en la Sección 4.2.7 se examina la portabilidad de rendimiento en diferentes implementaciones SYCL.

4.2.1. Rendimiento y funcionalidad

Inicialmente, se realizó una comparación del rendimiento de las distintas capacidades que ofrece la suite de SW#. El objetivo principal es evaluar y analizar el rendimiento en diferentes escenarios utilizando exclusivamente GPUs NVIDIA, para facilitar la comparación y garantizar la coherencia en los resultados. A continuación, se presentan las diferentes comparaciones de rendimientos entre CUDA y SYCL para la suite completa de SW#.

4.2.1.1. Tamaño del *work-group*

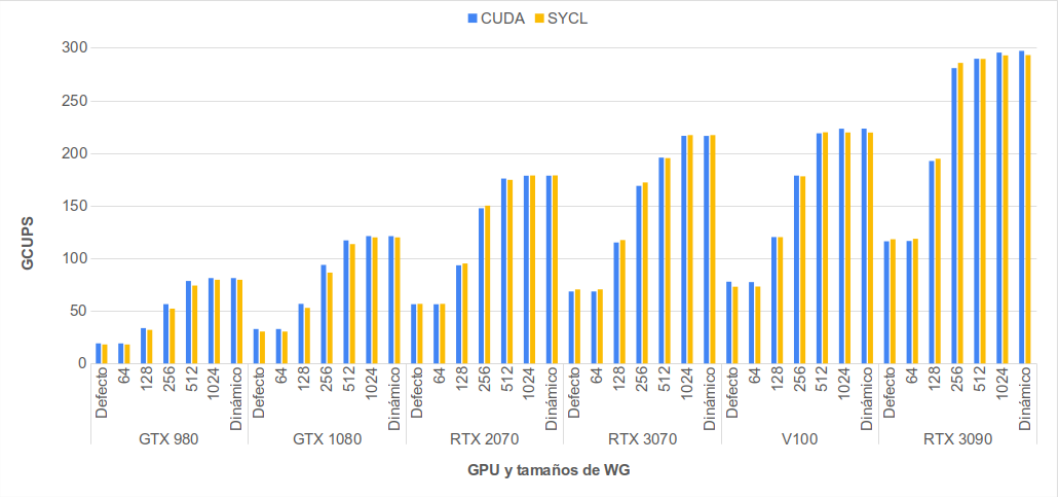

Figura 4.1: Comparación de rendimientos al variar el tamaño del WG.

La Figura 4.1 presenta el rendimiento de las versiones de CUDA y SYCL al variar el tamaño del WG, utilizando la base de datos Swiss-Prot y el algoritmo SW. Se puede observar que ambos códigos son sensibles al tamaño del WG. De hecho, la configuración dinámica logró los mejores resultados en todos los casos, aunque muy parecida a la configuración 1024. Además, es importante notar que ambos códigos son capaces de obtener más GCUPS al utilizar GPUs más potentes. Para los siguientes experimentos, se optó por una configuración dinámica de WG en ambos modelos de programación.

4.2.1.2. Longitudes de secuencia de consulta y bases de datos

La Figura 4.2 complementa la comparación anterior incluyendo la base de datos Env. NR, cuyo tamaño es aproximadamente 7 veces mayor que el de Swiss-Prot. Por un lado, se puede observar una pequeña pérdida de rendimiento al usar una base de datos más grande, que se va achicando (o incluso se revierte) a medida que se usa una GPU más potente. En particular, la diferencia es de 15 %, 13 %, 9 % y 2 % para la GTX 980, GTX 1080, RTX 2070 y RTX 3090, respectivamente. En cambio, en la V100 el uso de Env. NR alcanza 8 % más de GCUPS que con Swiss-Prot. Esta tendencia podría estar relacionada con la mayor velocidad y tamaño de los niveles de caché de las GPUs más recientes. Por otro lado, ningún código alcanzó el mejor rendimiento en todos los casos. La versión de CUDA mostró superioridad en la GTX 980 y la GTX 1080 para ambas bases de datos y también en la V100 y RTX 3090, pero solo para el caso de Swiss-Prot. Sin embargo, es importante señalar que la mejora del rendimiento es de hasta solamente un 2 % en el mejor de los casos. Una situación similar ocurre en la V100 y RTX 3090 con la base de datos Env. NR, donde la implementación en SYCL fue la más rápida, alcanzando hasta un 2 % más de GCUPS. Por último, la disparidad de rendimiento entre ambos códigos fue menor al 1 % en la RTX 2070 y la RTX 3070. Así, debido a las pequeñas diferencias de rendimiento, se puede concluir en que ambos modelos otorgan un rendimiento comparable al variar el tamaño de la base de datos.

La influencia de la longitud de la consulta puede observarse en la Figura 4.3 [6]. Por un

[6] Aunque se ejecutaron tanto los códigos de SYCL como los de CUDA, solo se incluye la versión de SYCL

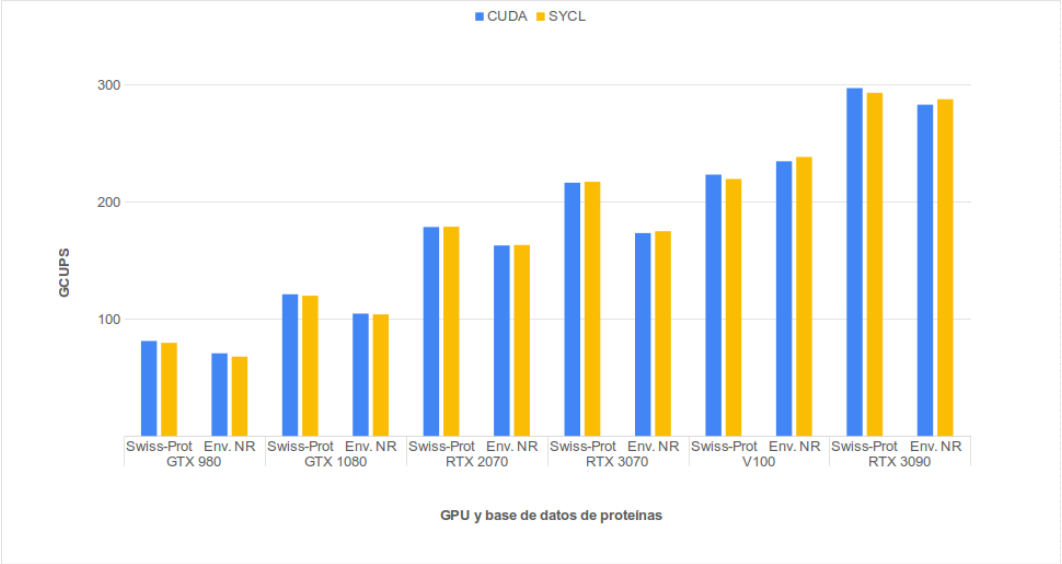

Figura 4.2: Comparación de rendimientos al variar la base de datos de proteínas.

lado, como se esperaba, una consulta más larga conduce a un mejor rendimiento. Por otro lado, este gráfico permite explorar de forma más detallada lo que se observó en la Figura 4.1, manifestando que aunque las GPU más potentes tienen un rendimiento superior, es necesario un volumen de trabajo suficientemente grande para aprovechar su potencia de cálculo. Por ejemplo, la RTX 3090 alcanza el mejor rendimiento, pero solo cuando la secuencia de la consulta supera los 3005 residuos.

4.2.1.3. Algoritmo de alineamiento y esquema de puntuación

Para evitar los sesgos de la configuración predeterminada, fueron considerados diferentes algoritmos de alineamiento y esquemas de puntuación para los mismos experimentos. Como muestran las Figuras 4.4 y 4.5, la diferencia de rendimiento para ambas variantes es del 2 % en promedio, aumentando hasta el 4 % en algunos casos (como por ejemplo, la V100). Por lo tanto, ninguno de estos parámetros parece tener un impacto en el rendimiento del código migrado. Esto demuestra que el código SYCL migrado no solo es funcional para las diferentes configuraciones de SW#, sino que también mantiene un rendimiento equiparable a CUDA en los distintos contextos.

4.2.1.4. Alineamiento de secuencias de ADN

El alineamiento por pares presenta diferentes desafíos de paralelización para la búsqueda de similitudes en bases de datos. SW# emplea el enfoque de paralelismo inter-tareas para el primero (*kernel* `swSolveSingle`) y el esquema de paralelismo intra-tarea para el segundo (*kernel* `swSolveShortGpu`). En consecuencia, la comparación de rendimiento en alineamientos de ADN se presenta en la Figura 4.6.

para mejorar la legibilidad del gráfico.

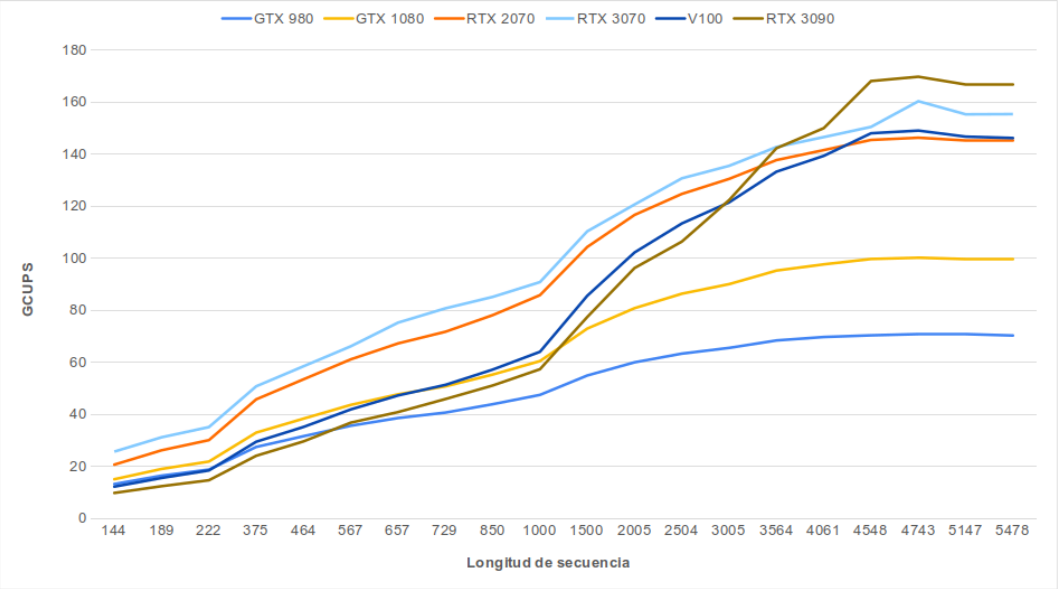

Figura 4.3: Comparación de rendimientos al variar la longitud de las secuencias.

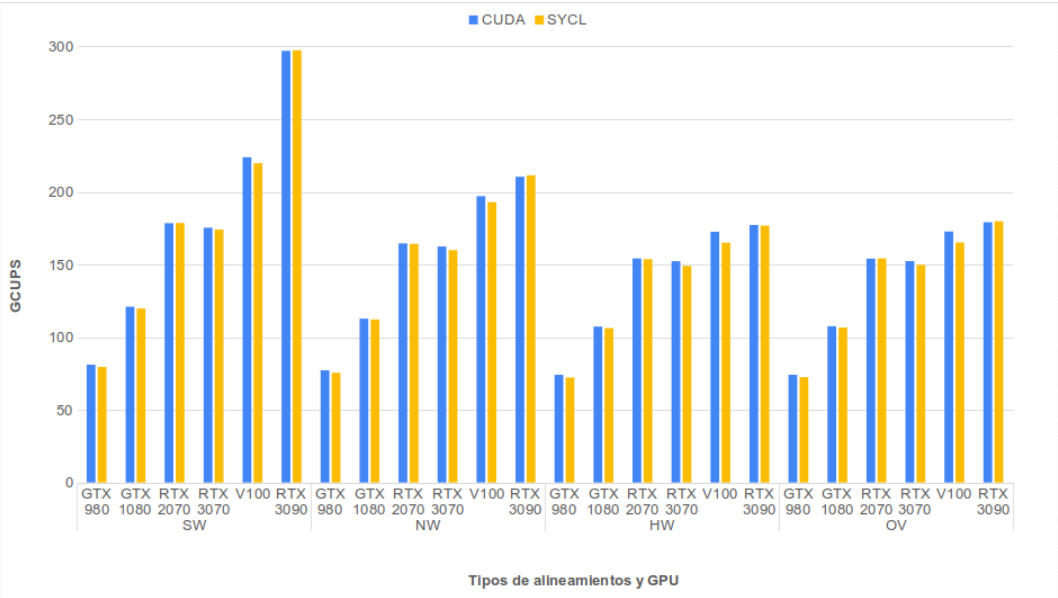

Figura 4.4: Comparación de rendimiento al variar el algoritmo de alineamiento.

Por un lado, a diferencia del caso de las proteínas, secuencias de ADN más largas no siempre conducen a más GCUPS. Este hecho se puede atribuir a las particularidades del alineamiento de secuencias de ADN, como el grado de similitud entre ellas, como ya se observó en [165]. Por otro lado, el rendimiento entre ambos modelos es similar, excepto en dos GPUs: en la RTX 2070, SYCL supera a CUDA en un 10 % en promedio, mientras que

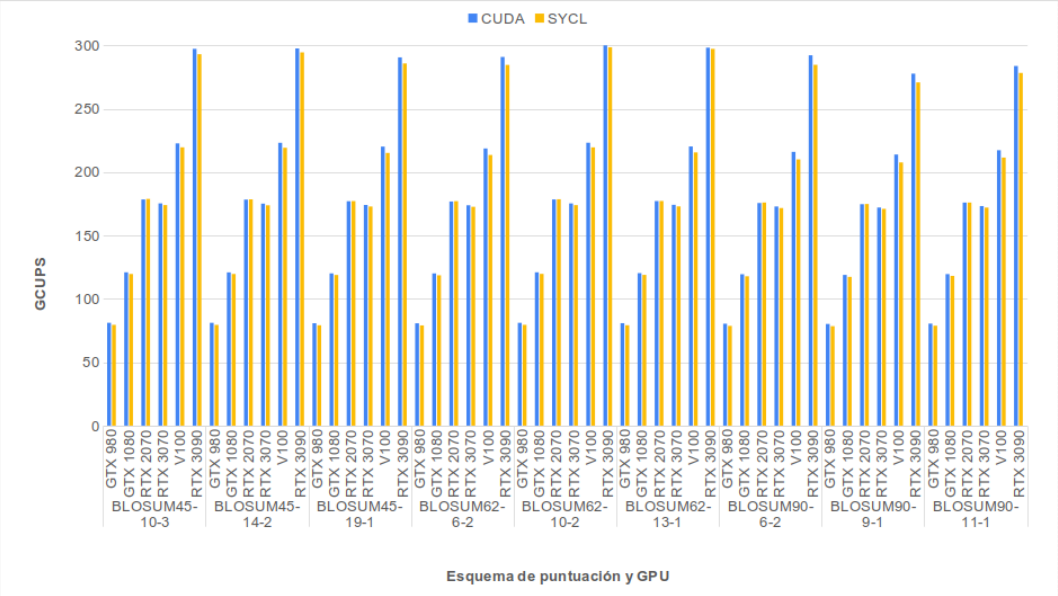

Figura 4.5: Comparación de rendimiento al variar la matriz de puntuación.

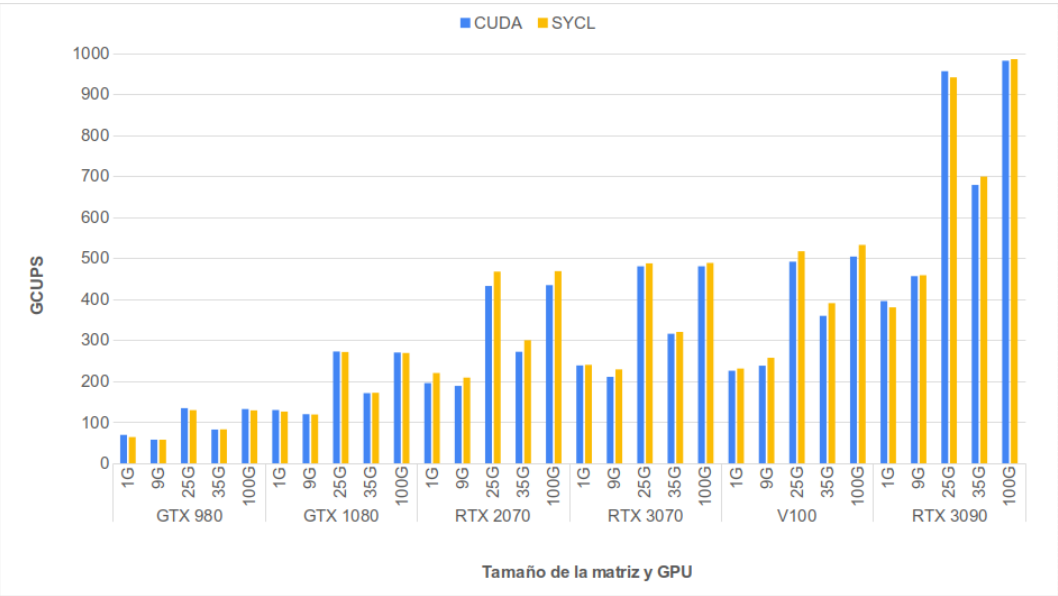

Figura 4.6: Comparación de rendimientos para alineamientos de ADN.

en la V100 la diferencia sigue siendo positiva pero ligeramente menor (7 %).

Para entender mejor sobre las causas de estas mayores diferencias de rendimiento, se realizó un perfilado de ambas ejecuciones de código en las GPUs RTX 2070 y RTX 3090 mediante el uso la herramienta NVIDIA Nsight Compute [166][7]. La Tabla 4.3 presenta algunas

[7]Las métricas de perfilado utilizadas se describen en [167].

Tabla 4.3: Perfiles de ejecuciones de alineamiento de secuencias de ADN (tamaño de matriz: 100G) para CUDA y SYCL en las GPU RTX 2070 y RTX 3090.

Sección	Métrica	RTX 2070			RTX 3090		
		CUDA	SYCL	SYCL/ CUDA Ratio	CUDA	SYCL	SYCL/ CUDA Ratio
Compute Workload Analysis	Executed Ipc Active (inst/cycle)	2.15	2.71	1.26	2.18	2.07	0.95
	Executed Ipc Elapsed (inst/cycle)	1.79	2.26	1.26	1.53	1.46	0.95
GPU Speed Of Light Throughput	Memory Throughput (%)	20.08	33.85	1.69	N/A	N/A	-
	DRAM Throughput (%)	0.16	0.20	1.25	0.42	0.42	1
	L1/TEX Cache Throughput (%)	24.18	40.74	1.68	25.72	24.43	0.95
	L2 Cache Throughput (%)	0.17	0.24	1.41	0.35	0.3	0.95
Memory Workload Analysis	Memory Throughput (Mbyte/sec)	677.46	853.17	1.26	3684.16	3503.77	0.95
	Max Bandwidth (%)	20.08	33.85	1.69	18.16	17.26	0.95
	Mem Pipes Busy (%)	20.08	33.84	1.69	18.16	17.26	0.95
Scheduler Statistics	Issued Warp Per Scheduler	0.54	0.68	1.26	N/A	N/A	-

métricas relevantes recopiladas de esta tarea experimental. Como se puede ver, SYCL supera a CUDA en varias métricas en la RTX 2070, no solo en gestión de memoria sino también en productividad computacional. Sin embargo, ambos códigos alcanzan valores prácticamente iguales en la RTX 3090. En este punto, se sospecha que podría estar relacionado con características particulares de la microarquitectura en contraste con el resto de ellos [8].

4.2.2. Modelo para portabilidad de rendimiento

Con el fin de evaluar la portabilidad de rendimiento, resulta necesario estimar el rendimiento teórico máximo del hardware para todas las GPUs y CPUs seleccionadas. Para ello se siguen las indicaciones detalladas en la Sección 2.4.2, lo cual requiere considerar tanto las características del hardware como las del algoritmo. Afortunadamente, el trabajo previo de Lan *et al.* [142] puede utilizarse como base para esta tarea. En ese artículo, la capacidad

[8] Ambas GPUs pertenecen a la misma microarquitectura de NVIDIA; a diferencia de generaciones anteriores, NVIDIA ha utilizado diferentes nombres en clave para cada segmento comercial (Turing es el nombre en clave para el segmento de consumidor mientras que Volta corresponde al segmento profesional)

de cómputo de diferentes dispositivos (incluyendo GPUs de NVIDIA, CPUs de Intel, y los discontinuados Intel Xeon Phis) se estimó utilizando la Ec.4.1:

$$Capability = Clock\_Rate \times Throughput \times Lanes \tag{4.1}$$

donde *Clock_Rate* se refiere a la frecuencia del reloj del dispositivo, *Throughput* se refiere a la cantidad de instrucciones que el dispositivo puede ejecutar en un ciclo de reloj y *Lanes* se refiere al número de carriles de la unidad de procesamiento vectorial (SIMD) del dispositivo. Luego, se debe contar el número de instrucciones emitidas en cada actualización de celda de la matriz de similitud. Así, en el contexto de alineamiento de secuencias, el rendimiento máximo teórico de cualquier dispositivo podría modelarse utilizando la Ec. 4.2:

$$Theo\_peak = \frac{Capability}{Instruction\_count\_one\_cell\_update} \tag{4.2}$$

Aunque este estudio solo considera las CPUs y GPUs, estas ecuaciones pueden servir como base para estimar su rendimiento máximo teórico en otros dispositivos, como las FPGAs. Para esta investigación, se adapta el modelo de rendimiento previo de [142] a las características del algoritmo SW# y también se extiende a otros proveedores de GPUs, como las GPUs de AMD e Intel (tanto discretas como integradas), y de CPUs de los mismos fabricantes. Las Tablas 4.4, 4.5 y 4.6, resumen el rendimiento máximo teórico de las GPUs seleccionadas de NVIDIA, Intel y AMD, respectivamente. Además, la Tabla 4.7 muestra el rendimiento para las CPUs. Ambas tablas utilizan la Ecuación 4.2 como referencia. En las restantes secciones de esta parte se proporciona información más detallada sobre los cálculos realizados.

4.2.2.1. Instrucciones *core* de SW#

De acuerdo con mediciones realizadas, $\tilde{9}9\,\%$ del tiempo de ejecución de SW# es destinado al cómputo de las matrices de similitud. Para dicho cómputo utiliza enteros de 32 bits y realiza 12 instrucciones por actualización de celda. El Algoritmo 1 presenta el fragmento de la actualización de celda en la matriz de similitud como en la Ec. 2.7, Ec. 2.8, y Ec. 2.9. Solo se requieren instrucciones de suma, resta y máximo para realizar una actualización de celda individual.

Algoritmo 1 Instrucciones *core* por actualización de celda en la matriz de similitud.

1: $E^1 = E_l - G_e$ {E_l: E de su vecino izquierdo}
2: $E^2 = H_l - G_o$ {H_l: H de su vecino izquierdo}
3: $E = \text{máx}(E^1, E^2)$
4: $F^1 = F_u - G_e$ {F_u: F de su vecino superior}
5: $F^2 = H_u - G_o$ {H_u: H de su vecino superior}
6: $F = \text{máx}(F^1, F^2)$
7: $H = H_{ul} + SM$ {H_{ul}: H de su vecino superior izquierdo}
8: $H = \text{máx}(H, E)$
9: $H = \text{máx}(H, F)$
10: $H = \text{máx}(H, 0)$
11: $A = H$ {A: una variable auxiliar}
12: $S = \text{máx}(H, S)$ {S: puntuación óptima}

4.2.2.2. Características arquitectónicas en las GPUs de NVIDIA

El término *# Cores* en una GPU de NVIDIA se refiere al número de SM. CUDA no sigue estrictamente un modelo de ejecución SIMD, pero adopta uno similar denominado modelo SIMT. Un *warp* está compuesto por un grupo de 32 hilos que ejecutan el mismo flujo de instrucciones. Según [142], "*un* `warp` *en SIMT es equivalente a un vector en SIMD, y un* `hilo` *en SIMT es equivalente a una* `vector lane` *en SIMD*". El *throughput* de instrucciones depende de la Capacidad de Cómputo CUDA (CC) de cada GPU de NVIDIA [9].

4.2.2.3. Características arquitectónicas en las GPUs de AMD

En la arquitectura GCN5, el término *# Cores* representa el número de CUs. AMD denomina *wavefront* y WI a los equivalentes del *warp* e hilo de NVIDIA, respectivamente. En GCN5, el tamaño de *wavefront* es fijo, siendo su valor igual a 64. Cada CU contiene una unidad vectorial SIMD32, siendo capaz de computar 64 instrucciones de suma/resta/máximo por ciclo (`Int32`). Esto significa que el *throughput* de instrucciones es 1 para cada WI.

En la arquitectura RDNA2, el término *# Cores* también representa el número de CUs, aunque estos están agrupados de a pares en Procesadores de WG. A diferencia de GCN5, RDNA2 admite tamaños de *wavefront* tanto de 32 como de 64 WI, aunque se prioriza el primero. Cada CU contiene dos unidades vectoriales SIMD32, siendo capaces de computar 64 instrucciones de suma/resta/máximo por ciclo (`Int32`). Esto significa que el *throughput* de instrucciones es 2 para cada WI.

4.2.2.4. Características arquitectónicas en las GPUs de Intel

En el segmento discreto, Intel tiene una filosofía de diseño de GPU bastante diferente a la de NVIDIA y AMD. El bloque fundamental de la microarquitectura Intel Xe es el Núcleo Xe, cada uno de los cuales tiene 16 Motores Vectoriales Xe (XVEs, por sus siglas en inglés) [10] que pueden ejecutar 8 instrucciones de suma/resta/máximo por ciclo (`Int32`). Por lo tanto, los Núcleos Xe y los XVEs se corresponden con *# Cores* y *# Lanes*, respectivamente, en el modelo propuesto.

En el segmento integrado, tanto las microarquitecturas Gen9 como Gen12 son similares desde una perspectiva de diseño, diferenciándose principalmente en la cantidad de recursos computacionales. En estas microarquitecturas, el bloque fundamental es el Subsegmento, cada uno de los cuales tiene 8 EUs capaces de ejecutar 8 instrucciones de suma/resta/máximo por ciclo (`Int32`). Así, los Subsegmentos y las EUs se refieren a *# Cores* y *# Lanes*, respectivamente, en el modelo propuesto.

4.2.2.5. Características arquitectónicas en CPUs

Tanto para CPUs de Intel como de AMD, el término *# Cores* representa el número de núcleos del procesador, cada uno capaz de ejecutar instrucciones de manera independiente, siendo fundamental para el rendimiento en aplicaciones paralelas. El *# Lanes* se define por la cantidad de Unidades de Procesamiento Vectorial (VPUs, por sus siglas en inglés) y su ancho vectorial/conjunto de instrucciones SIMD soportado (como SSE, AVX, AVX-512), que permite a un núcleo realizar múltiples operaciones simultáneamente. El *# throughput* de instrucciones depende de la operación a realizar. Como el uso intensivo de las VPUs lleva a una mayor disipación de calor y consumo de energía, los procesadores modernos

[9] `https://docs.nvidia.com/cuda/cuda-c-programming-guide/#maximize-instruction-throughput`

[10] También conocidos como Unidades de Ejecución (EUs, por sus siglas en inglés).

incorporan diversas tecnologías que llevan a reducir la frecuencia de reloj para contrarrestar su impacto [168]. Por esta razón, la frecuencia operativa cuando todos los núcleos realizan cómputo intensivo nunca es la frecuencia máxima, siendo habitualmente la base o alguna un poco más alta.

Para el caso de arquitecturas híbridas de CPU (como la Alder Lake de Intel) [169], se debe hacer una salvedad. Como en estas arquitecturas hay dos clases de núcleos (*P-cores* y *E-cores*), el máximo rendimiento teórico se obtiene mediante la suma de los rendimientos máximos parciales.

4.2.2.6. Rendimiento teórico máximo para plataformas usadas

Para estas pruebas de portabilidad de rendimiento, se utilizaron GPUs y CPUs de distintos fabricantes. Las Tablas 4.4, 4.5, 4.6 y 4.7 muestran las características y los picos de GCUPS para las GPUs de NVIDIA, Intel, AMD y las CPUs utilizadas, respectivamente.

Tabla 4.4: Especificaciones de las GPUs de NVIDIA y su rendimiento máximo teórico en términos de GCUPS.

Fabricante	NVIDIA					
Modelo	GTX 980	GTX 1080	RTX 2070	V100	RTX 3070	RTX 3090
Año	2014	2016	2018	2017	2020	2020
Tipo	Discreta					
Microarq.	Maxwell (CC 5.2)	Pascal (CC 6.1)	Turing (CC 7.5)	Volta (CC 7.0)	Ampere (CC 8.6)	Ampere (CC 8.6)
# Cores	16	20	36	80	46	82
# Lanes	32					
Throug. inst.	4/2			2		
Reloj (MHz)	1216	1733	1620	1380	1725	1695
Pico teórico (GCUPS)	155.64	277.28	311.04	588.80	423.20	741.20

El rendimiento de instrucciones para GTX 980 y GTX 1080 es de 4 para suma/resta y 2 para máximo/mínimo. Las instrucciones *core* incluyen 5 suma/resta y 6 máximo. Por lo tanto, el rendimiento equivalente es 3.
La cantidad de instrucciones *core* para cada actualización de celda es de 12.

4.2.3. Portabilidad en GPU individual

Se llevó a cabo una comparación inicial entre el rendimiento de CUDA y SYCL en las GPUs de NVIDIA, detallada en la Figura 4.7. Tal como se puede apreciar, ambos modelos de programación logran obtener rendimientos de GCUPS prácticamente idénticos. Por un lado, se pudo observar la mayor diferencia de rendimiento a favor de SYCL en la Tesla V100 (3.4 %). Por otro lado, ocurre lo contrario en la GTX 980, donde CUDA supera a SYCL en

Tabla 4.5: Especificaciones de las GPUs de Intel y su rendimiento máximo teórico en términos de GCUPS.

Fabricante	Intel		
Modelo	Intel Arc 770	UHD 630	UHD 770
Tipo	Discreta	Integrada	
Microarq.	Xe HPG	Gen9.5	Gen12.2
# Cores	32	3	4
# Lanes	16	8	
Throug. inst.	8		
Reloj (MHz)	2400	1200	1650
Pico teórico (GCUPS)	819.2	19.2	35.2

Tabla 4.6: Especificaciones de las GPUs de AMD y su rendimiento máximo teórico en términos de GCUPS.

Fabricante	AMD	
Modelo	RX 6700 XT	Radeon Vega 6
Tipo	Discreta	Integrada
Microarq.	RDNA 2	GCN5
# Cores	40	6
# Lanes	32	64
Throug. inst.	2	1
Reloj (MHz)	2581	1100
Pico teórico (GCUPS)	550.61	35.20

Tabla 4.7: Especificaciones de las CPUs de Intel y AMD y su rendimiento máximo teórico en términos de GCUPS.

Fabricante	Intel							AMD
Modelo	Core i5-7400	Core i5-10400F	Xeon E5-1620 v3	Xeon E5-2695 V3	Xeon Gold 6138	Core i9-9900K	Core i9-13900K	Ryzen 3
Segmento	Escritorio		Servidor			Escritorio		*Mobile*
Microarq.	Kaby Lake	Comet Lake	Sandy Bridge-E	Haswell	Skylake	Coffee Lake-R	Raptor Lake-S	Lucienne
# Cores	4	6	4	14	40	8	8/16	4
# Lanes	8		32			8		
Throug. inst.	1							
Reloj (MHz)	3300	4000	3500	1900	1900	4700	3000/ 2200	3600
Pico (GCUPS)	8.80	16.00	9.33	35.47	101.33	25.07	39.47 [a]	9.60

[a] La CPU Intel Core i9-13900K presenta una arquitectura heterogénea, integrando núcleos de alto rendimiento (P-cores) diseñados para tareas de cómputo secuencial, y núcleos eficientes (E-cores) enfocados en la ejecución paralela. Ambos tipos de núcleos son compatibles con la tecnología AVX2. A diferencia de los otros casos, no se encontró información específica sobre la frecuencia a la que trabajan bajo condiciones de vectorización con AVX2 o en situaciones de turbo para todos los núcleos. Por ende, se consideró la frecuencia base como una referencia, aunque es probable que las velocidades reales sean ligeramente superiores.

un 4.6 %. A partir de este análisis, se concluye que tanto CUDA como SYCL son capaces de ofrecer un rendimiento comparable para este caso de estudio en las GPUs de NVIDIA.

La comparación detallada del rendimiento y la eficiencia arquitectónica de los códigos CUDA y SYCL en las GPUs de NVIDIA, AMD e Intel se presenta en la Tabla 4.8. En esta tabla se muestra el rendimiento teórico máximo, el rendimiento logrado tanto para CUDA como para SYCL, y la eficiencia arquitectónica correspondiente para cada plataforma.

En las GPUs de NVIDIA, tanto CUDA como SYCL obtuvieron rendimientos y eficiencias comparables, como se mencionó previamente en el análisis de la Figura 4.7. Las GPUs más potentes lograron valores más altos de GCUPS, lo cual era de esperar. En cuanto a la eficiencia arquitectónica, los valores se encuentran en el rango del 37 % al 52 %. Es importante destacar que, aunque la GPU RTX 3090 presenta el valor más alto de GCUPS, la más eficiente resulta ser la GPU RTX 2070. En el caso de las GPUs de AMD e Intel, solo se muestran los resultados para SYCL, ya que CUDA solo es compatible con las GPUs de NVIDIA. Esto resalta la mayor portabilidad de SYCL sobre CUDA, como se mencionó anteriormente.

Los resultados obtenidos de la versión SYCL en estas GPUs son en gran medida satisfactorios. En primer lugar, la eficiencia arquitectónica de SYCL en la dGPU de AMD (51.7 %) es prácticamente igual a la mejor tasa de eficiencia lograda en las GPU de NVIDIA (52.4 %). Por otro lado, SYCL supera ampliamente esta marca en las 2 iGPUs, logrando valores de eficiencia arquitectónica de hasta +23.1 %. Los únicos aspectos negativos se encuentran en los resultados de la dGPU Arc A770 de Intel y la iGPU Vega 6 de AMD, donde la eficiencia arquitectónica cae al 23.3 % y 21.3 %, respectivamente. Este valor representa el rendimiento

Tabla 4.8: GCUPS y eficiencias arquitectónicas de los códigos CUDA y SYCL en GPUs individuales.

Plataforma			**CUDA**		**SYCL**	
Fabricante	**GPU**	**GCUPS pico**	**GCUPS alcanzado**	**Eficiencia arq.**	**GCUPS alcanzado**	**Eficiencia arq.**
NVIDIA	GTX 980	155.5	70.6	45.3 %	67.7	43.5 %
	GTX 1080	277.2	104.5	37.7 %	103.8	37.4 %
	RTX 2070	311.0	162.5	52.2 %	163.1	52.4 %
	Tesla V100	588.8	224.9	38.2 %	233.0	39.5 %
	RTX 3070	423.2	173.1	40.9 %	174.4	41.2 %
	RTX 3090	741.3	280.2	37.8 %	288.6	38.9 %
Intel	Arc A770	819.2	×	NA	191.4	23.3 %
	UHD 630	19.2	×	NA	13.1	68.4 %
	UHD 770	35.2	×	NA	26.6	75.7 %
AMD	RX 6700 XT	550.6	×	NA	284.4	51.7 %
	Vega 6	35.20	×	NA	7.5	21.3 %

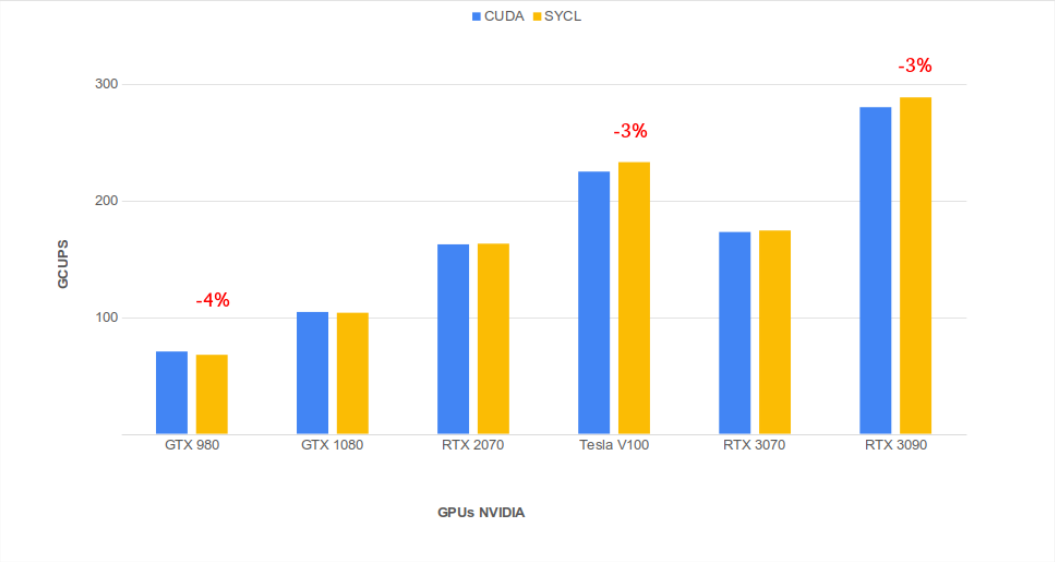

Figura 4.7: Comparación de rendimiento entre CUDA y SYCL en GPUs individuales de NVIDIA.

más bajo y la causa podría estar relacionada con la filosofía de diseño de estas GPUs. No obstante, sería necesario realizar un perfilado del código para obtener más información sobre la causa.

Tabla 4.9: Portabilidad del rendimiento de los códigos CUDA y SYCL en GPUs individuales.

	$\Phi(\alpha, p, H)$	
Conjunto de plataformas (H)	**CUDA**	**SYCL**
NVIDIA	42 %	42.2 %
AMD (discreto)	NA	51.7 %
AMD (integrado)	NA	21.3 %
AMD (todos)	NA	36.5 %
Intel (discreto)	NA	23.3 %
Intel (integrado)	NA	72.0 %
Intel (todos)	NA	55.8 %
NVIDIA ∪ AMD	NA	40.7 %
NVIDIA ∪ Intel	NA	47.2 %
Intel ∪ AMD	NA	48.1 %
NVIDIA ∪ AMD ∪ Intel	NA	44.8 %

La evaluación de la portabilidad del rendimiento de los códigos CUDA y SYCL se presenta en la Tabla 4.9. En esta tabla se puede apreciar que los resultados agregados son coherentes con los observados de forma individual previamente. En el caso de las GPUs de NVIDIA, se observa que la portabilidad del rendimiento de ambos modelos de programación es bastante similar, con valores del 42 % y 42.2 %, respectivamente. Esto confirma lo mencionado anteriormente, indicando que ambos modelos pueden ofrecer un nivel consistente de rendimiento en las diferentes GPUs de NVIDIA utilizadas en las pruebas.

Por otro lado, en el caso de las GPUs de Intel, se encontró que SYCL demostró valores de eficiencia arquitectónica bastante favorables en las iGPUs, a diferencia de la menor eficiencia exhibida en la dGPU. Es interesante resaltar que ocurre lo contrario en AMD, donde se logra una buena eficiencia arquitectónica en la dGPU pero no así en la iGPU. Al considerar la combinación de GPU de AMD e Intel, se observa que SYCL logra la mayor portabilidad del rendimiento del segmento intermedio de la tabla. Sin embargo, la portabilidad de rendimiento disminuye cuando también se incluyen las GPUs de NVIDIA (último segmento), ya que el rendimiento de SYCL es menor en estos dispositivos.

En relación con el análisis anterior, se puede concluir en que SYCL supera consistentemente a CUDA en términos de portabilidad del rendimiento en este estudio. Para ser más precisos, SYCL logró casi la misma eficiencia arquitectónica que CUDA considerando 6 GPU de NVIDIA con 5 microarquitecturas diferentes. Además, SYCL no solo fue capaz de ejecutarse en GPU de varios proveedores (AMD e Intel), sino que también mostró una eficiencia arquitectónica superior en 3 de los 5 casos probados. Esto demuestra no solo la extensa compatibilidad de SYCL, sino también su capacidad para mejorar el rendimiento en una amplia gama de GPUs para esta aplicación.

4.2.4. Portabilidad en multi-GPU

Con el fin de complementar el análisis anterior de GPUs individuales, se realizó una comparación de rendimiento entre CUDA y SYCL utilizando distintas combinaciones de múltiples GPUs NVIDIA (ver Fig. 4.8). Al igual que en el caso de GPUs individuales, los dos modelos de programación logran valores prácticamente iguales de GCUPS cuando se utilizan dispositivos NVIDIA, tanto para configuraciones homogéneas como heterogéneas de múltiples GPUs. Mientras que CUDA supera a SYCL cuando se utiliza 2×GTX1080 en aproximadamente un 1 %, SYCL logra mejor rendimiento en todos los demás casos, alcanzando hasta un 5 % más de GCUPS. Por lo tanto, es posible determinar que SYCL no implica una sobrecarga adicional cuando se utilizan múltiples GPUs.

La Tabla 4.10 presenta una comparación más detallada del rendimiento y la eficiencia arquitectónica de los códigos CUDA y SYCL en 5 configuraciones diferentes de múltiples GPUs. Se puede observar que para múltiples GPUs NVIDIA, las tasas de eficiencia logradas al utilizar 2 GPUs combinadas son un poco más bajas que cuando se utiliza una sola GPU. Este comportamiento ocurre en 3 de las 4 configuraciones probadas (la excepción es cuando se utiliza 2×Tesla V100) y puede explicarse por 2 razones: por un lado, es habitual que la eficiencia disminuya al fijar el tamaño del problema y aumentar la cantidad de recursos computacionales; por otro lado, la estrategia de distribución de carga de trabajo de *SW#* es muy sencilla, ya que distribuye las secuencias de consulta entre las GPUs y no considera la potencia de cómputo de cada una. Debido a que estas secuencias no tienen la misma longitud, puede producirse un desequilibrio de carga entre las GPUs, lo que reduce el rendimiento.

Finalmente, SYCL demuestra una vez más su mayor portabilidad funcional con el caso de múltiples GPUs de Intel. A pesar de que el rendimiento no es óptimo debido a las razones previamente mencionadas, es relevante destacar la habilidad de SYCL para ejecutar simultáneamente 2 GPUs de distinto tipo (una iGPU y una dGPU).

4.2.5. Portabilidad en CPU

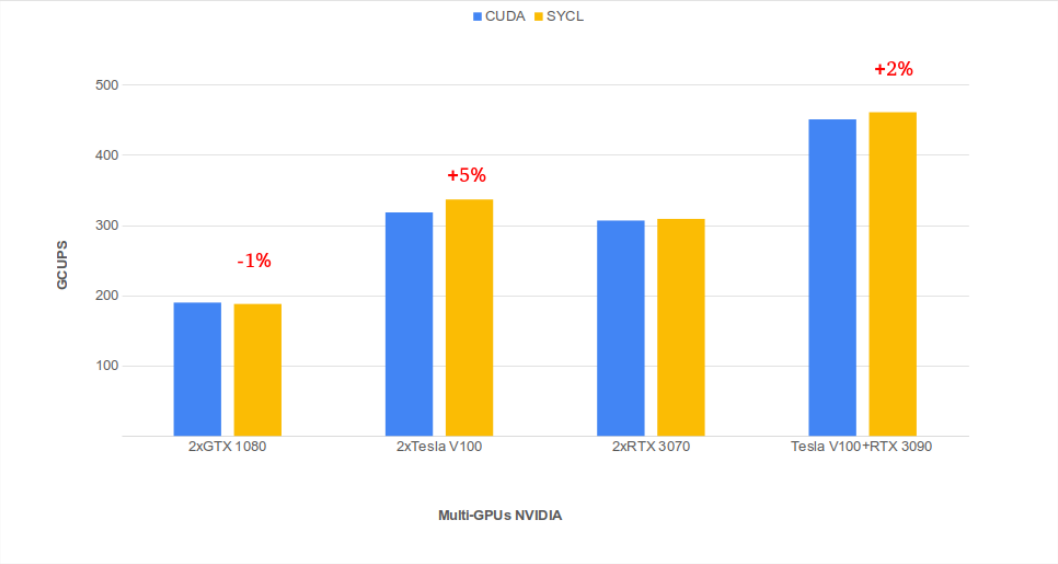

Figura 4.8: Comparación de rendimiento entre CUDA y SYCL en múltiples GPUs de NVIDIA.

La Figura 4.9 ilustra una comparación de rendimiento entre una GPU de NVIDIA (RTX 980) frente a varias CPUs de Intel y una CPU de AMD. En realidad, el propósito de esta figura no es comparar el rendimiento, sino mostrar la portabilidad funcional del código SYCL a diferentes arquitecturas de CPU. A pesar de las diferencias arquitectónicas inherentes entre las GPUs y CPUs, la adaptabilidad del código SYCL se manifiesta en su capacidad de funcionar correctamente en una amplia gama de CPUs, incluso de diferentes fabricantes, incluyendo modelos como Intel Core i5-7400, AMD Ryzen 3, entre otros. Es importante remarcar que incluso fue posible ejecutar en la CPU Core i9-13900K, la cual se basa en una arquitectura híbrida de CPU.

En todos los casos anteriores, los resultados fueron correctos. Aunque originalmente diseñado para GPU, el código SYCL se pudo adaptar y ejecutar en CPUs de diferentes fabricantes sin cambios significativos. En ese sentido, resulta importante destacar dos aspectos: (1) la ejecución de estas pruebas requirió solamente cambiar el *backend* al momento de compilar; (2) dado que la versión de DPC++ portada es código SYCL puro, la ejecución de esta versión en otras arquitecturas diferentes sólo necesitará de un compilador compatible.

Aunque la portabilidad funcional a CPUs fue comprobada, resulta interesante analizar la correspondiente portabilidad de rendimiento. Es por lo que la Tabla 4.11 muestra el rendimiento en GCUPS y la eficiencia arquitectónica del código SYCL ejecutado en CPUs usando ambos modelos de programación. Aunque CUDA no es aplicable en CPUs, esta columna se incluye para enfatizar la compatibilidad exclusiva de SYCL en este contexto. De todas las CPUs, el Core i5-10400F sobresale con una eficiencia máxima lograda de 51.2 %. En sentido opuesto, el Xeon Gold 6138 es el que menor eficiencia arquitectónica ofrece, a pesar de contar con el mayor pico teórico. Aunque las CPUs ofrecen menos GCUPS en comparación con las GPUs, la eficiencia arquitectónica supera el 38 % en 7 de los 8 casos.

Para profundizar el análisis anterior, la Tabla 4.12 presenta la portabilidad de rendimiento en CPUs discriminada por fabricantes y segmentos. En cuanto a CPUs de Intel, el segmento de escritorio pareciera tener una pequeña ventaja sobre el de servidor. Una ten-

Tabla 4.10: GCUPS y eficiencias arquitectónicas de los códigos CUDA y SYCL en múltiples GPUs.

Plataforma			CUDA		SYCL	
Fabricante	**GPUs**	**GCUPS pico**	**GCUPS obt.**	**Arq efi.**	**GCUPS obt.**	**Arq efi.**
NVIDIA	2× GTX 1080	554.6	189.8	34.2 %	187.8	33.9 %
	2× Tesla V100	846.4	318.1	27.0 %	336.5	28.6 %
	2× RTX 3070	1177.6	306.5	36.2 %	308.9	36.5 %
	Tesla V100 ∪ RTX 3090	1330.1	450.5	33.8 %	460.7	34.6 %
Intel	Arc A770 ∪ UHD 770	854.4	×	NA	126.8	14.8 %

dencia parecida ocurre al separar por fabricantes, siendo Intel levemente superior frente a AMD. Aunque esta última cuestión podría estar relacionada con el hecho de que el ecosistema oneAPI es desarrollado por Intel (y por ende, los compiladores podrían producir código más eficiente para sus propios procesadores), es importante tener en cuenta que cada conjunto de plataformas cuenta con una pequeña cantidad de CPUs y se necesitarían más de ellas para poder generalizar.

Finalmente, la métrica $\bar{\Phi}$ permite evaluar qué tan portable es el rendimiento de una aplicación al cambiar de arquitectura. En ese sentido, se debe tener en cuenta que SW# es un código CUDA específicamente diseñado para GPUs NVIDIA y su equivalente SYCL alcanzó un valor de $\bar{\Phi}$ de 42.2 % al considerar 6 de ellas (con 5 microarquitecturas diferentes). Se puede observar en la Tabla 4.12 que la portabilidad de rendimiento combinando todos los segmentos y fabricantes de CPUs fue del 41.39 %, siendo la diferencia menor al 1 %. Este hecho resalta la capacidad del código SYCL para ejecutarse eficientemente en múltiples plataformas y su ventaja en términos de portabilidad, lo cual resulta indispensable en ambientes donde la disponibilidad de hardware es diversa y la elección del mismo es flexible.

4.2.6. Portabilidad en CPU-GPU

La Figura 4.10 muestra una comparación de rendimiento en GCUPS entre diferentes combinaciones híbridas de CPU-GPU. Es importante resaltar que el objetivo de este experimento no está en comparar el rendimiento, sino en demostrar la portabilidad del código SYCL. En esta figura, se pueden apreciar las diferencias de rendimiento en una variedad de dispositivos de NVIDIA, Intel y AMD, incluyendo combinaciones de CPU Intel + GPU de NVIDIA, CPU Intel + dGPU Intel, CPU Intel + dGPU AMD, CPU AMD + iGPU

Tabla 4.11: GCUPS y eficiencias arquitectónicas de los códigos CUDA y SYCL en CPUs individuales.

Plataforma			**CUDA**		**SYCL**	
Fabricante	**CPUs**	**GCUPS pico**	**GCUPS obt.**	**Arq efi.**	**GCUPS obt.**	**Arq efi.**
Intel	Core i5-7400	8.8	×	NA	4.2	47.6 %
	Core i5-10400F	16.0	×	NA	8.2	51.2 %
	Xeon E5-1620	9.3	×	NA	3.9	42.1 %
	Xeon E5-2695	35.5	×	NA	15.8	44.6 %
	Xeon Gold 6138	101.3	×	NA	28.5	28.2 %
	Core i9-9900K	25.1	×	NA	9.6	38.4 %
	Core i9-13900K	39.47	×	NA	15.9	40.2 %
AMD	Ryzen 3	9.6	×	×	3.7	38.8 %

Tabla 4.12: Portabilidad del rendimiento de los códigos CUDA y SYCL en CPUs individuales.

	$\bar{\Phi}(\alpha, p, H)$	
Conjunto de plataformas (H)	**CUDA**	**SYCL**
Intel CPUs (escritorio)	NA	44.35 %
Intel CPUs (servidor)	NA	38.30 %
AMD CPU (mobile)	NA	38.82 %
Intel CPUs	NA	41.76 %
AMD CPU	NA	38.82 %
Intel ∪ AMD	NA	41.39 %

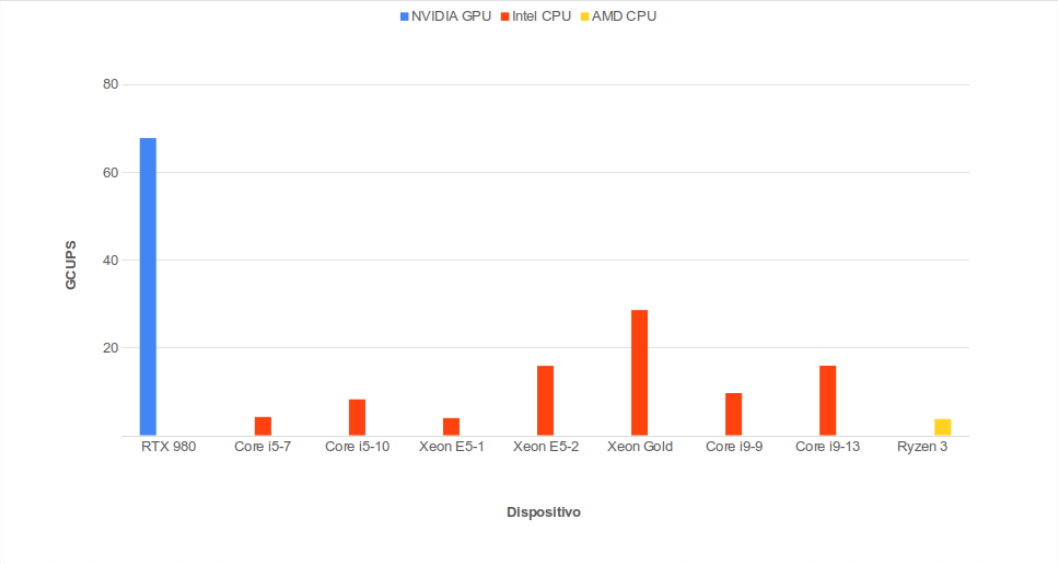

Figura 4.9: Comparación de rendimiento entre CPUs Intel y CPU AMD, usando de referencia GPU NVIDIA.

AMD, e incluso CPU Intel + dGPU Intel + iGPU Intel. En particular, se puede destacar la combinación de RTX3090 junto con Xeon Gold 6138 por su alto rendimiento, indicativo de una combinación efectiva entre las capacidades de cómputo paralelo de la GPU y la robustez de la CPU.

Por otro lado, la Tabla 4.13 proporciona una vista detallada de los GCUPS y las eficiencias arquitectónicas de los códigos ejecutados en múltiples combinaciones CPU-GPU utilizando SYCL. Como en los casos anteriores, se observa que CUDA no es aplicable en el contexto de CPUs, lo cual refuerza la relevancia de SYCL en estos escenarios de cómputo híbrido. Analizando la tabla, se aprecia que las combinaciones que involucran la GPU GTX980 (Maxwell) y RTX3070 (Ampere) muestran el rendimiento más alto, resaltando la escalabilidad del código en diferentes arquitecturas NVIDIA. Sin embargo, incluso las combinaciones con menores GCUPS, como la VEGA6 y RYZEN 3, demuestran la portabilidad funcional en una gama más amplia de hardware, incluyendo aquellos con recursos más limitados.

La eficiencia arquitectónica de SYCL en diferentes combinaciones CPU-GPU varía, reflejando la adaptabilidad del código a las particularidades de cada arquitectura. En la parte superior de la tabla, las combinaciones de NVIDIA con Intel CPUs tienden a mostrar una mayor eficiencia arquitectónica, lo que sugiere una integración más efectiva entre estas dos tecnologías. En contraste, las combinaciones de Intel GPU y CPU, así como la de AMD CPU y GPU, presentan eficiencias arquitectónicas menores, lo que puede indicar una oportunidad para optimizar la sinergia entre estas arquitecturas. La Tabla 4.14 presenta la portabilidad de rendimiento para cada combinación híbrida. En esta tabla se puede observar que el rendimiento más alto para SYCL se obtiene al utilizar una CPU de Intel y una GPU de NVIDIA, alcanzando un 27.39 %, lo cual podría indicar una optimización más efectiva o una mejor compatibilidad entre estos dispositivos para el procesamiento de estos códigos. Por el contrario, el rendimiento más bajo se observa al utilizar una CPU de Intel y una dGPU de AMD, con un 5.97 %. Además, se puede notar que las configuraciones que involucran iG-

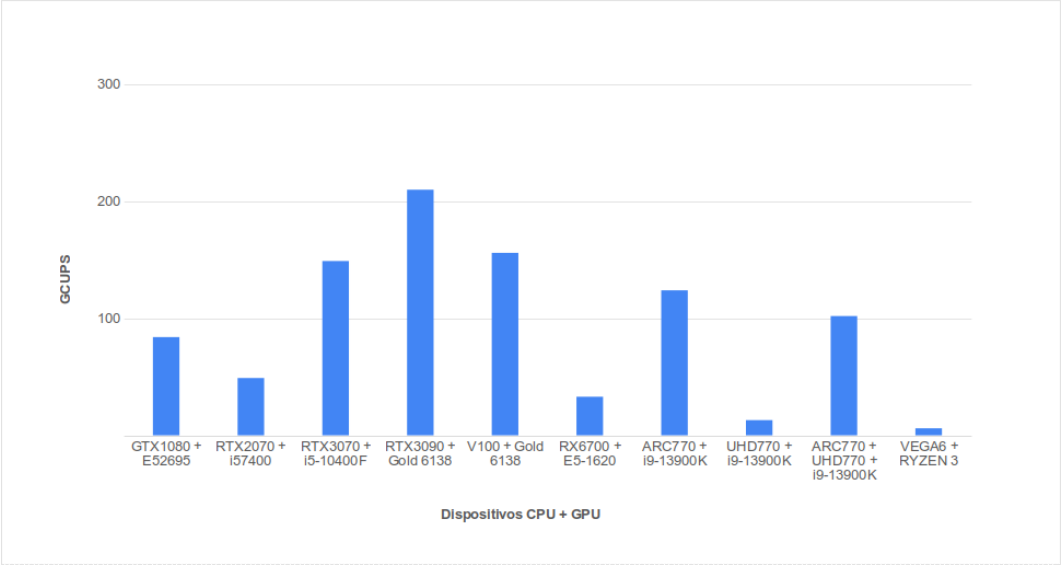

Figura 4.10: Comparación de rendimiento en combinaciones CPU-GPU.

PUs de AMD e Intel presentan rendimientos intermedios, fluctuando entre el 14.03 % y el 17.80 %, respectivamente. En general, estos resultados indican que la elección del tipo de CPU y GPU puede tener un impacto significativo en el rendimiento de los códigos SYCL, lo cual es una consideración importante para la optimización de aplicaciones en entornos que integran CPU+GPU. Sería conveniente contar con un conjunto más amplio de GPUs (especialmente de Intel y AMD) y de CPUs (especialmente de AMD) para poder robustecer las observaciones realizadas.

Finalmente, y de manera general, se observa que el rendimiento se ve afectado negativamente al introducir dispositivos mucho más lentos que las GPUs, como es el caso de las CPUs en esta situación. Así como sucede en los experimentos multi-GPU, los resultados se ven influenciados principalmente por la estrategia de distribución de carga de trabajo de SW#, la cual no considera las capacidades de los dispositivos. Esto puede ocasionar que la CPU deba realizar alineamientos de secuencias más extensas que las asignadas a la GPU, lo cual tiene un impacto negativo en el rendimiento final. Además, aunque no es equivalente a utilizar dos GPUs simultáneamente, la combinación de CPU+GPU también se ve afectada al fijar el tamaño del problema y aumentar la cantidad de recursos computacionales. Este escenario da lugar a posibles optimizaciones futuras.

4.2.7. Portabilidad en implementaciones SYCL

Para verificar la portabilidad entre diferentes implementaciones de SYCL, se decidió compilar y ejecutar el código portado en varias GPUs y CPUs utilizando el framework AdaptiveCpp. Para esta tarea, el paso de estandarización SYCL de la Sección 3.4.5 resultó fundamental. La Figura 4.11 presenta el rendimiento logrado para las versiones de oneAPI y AdaptiveCpp al utilizar la base de datos Swiss-Prot. Se pueden observar algunas pérdidas de rendimiento en AdaptiveCpp cuando se emplea el compilador *genérico* en lugar del *específico para el objetivo* (hasta un 15 %). En la dirección opuesta, no se pueden notar di-

Tabla 4.13: GCUPS y eficiencias arquitectónicas de los códigos CPU-GPU en múltiples combinaciones.

Comb.	Plataforma		CUDA		SYCL	
	GPU U CPU	**GCUPS pico**	**GCUPS obt.**	**Arq efi.**	**GCUPS obt.**	**Arq efi.**
NVIDIA U Intel CPU	GTX980 + E52695	191	×	NA	73	38.1 %
	GTX1080 + E52695	287	×	NA	84	29.5 %
	RTX2070 + i57400	320	×	NA	49	15.3 %
	RTX3070 + i5-10400F	439	×	NA	149	33.9 %
	RTX3090 + Gold 6138	843	×	NA	210	24.9 %
	V100 + Gold 6138	690	×	NA	156	22.6 %
Intel GPU U Intel CPU	ARC770 + i9-13900K	859	×	NA	124	14.5 %
	UHD770 + i9-13900K	75	×	NA	13	17.8 %
	ARC770 + UHD770 + i9-13900K	894	×	NA	128	14.3 %
AMD GPU U Intel CPU	RX6700 + E5-1620	560	×	NA	33	6.0 %
AMD GPU U Intel CPU	VEGA6 + RYZEN 3	45.0	×	NA	6	14.0 %

Tabla 4.14: Portabilidad del rendimiento de los códigos CUDA y SYCL en CPU+GPU.

Conjunto de plataformas (H)	$\bar{\Phi}(\alpha, p, H)$ **CUDA**	**SYCL**
Intel CPU ∪ NVIDIA GPU	NA	27.39 %
Intel CPU ∪ AMD dGPU	NA	5.97 %
AMD CPU ∪ AMD iGPU	NA	14.03 %
Intel CPU ∪ Intel iGPU	NA	17.80 %
Intel CPU ∪ Intel dGPU	NA	14.49 %
Intel CPU ∪ Intel GPU	NA	14.57 %
Intel CPU∪ GPU	NA	22.51 %
CPU∪ GPU	NA	21.66 %

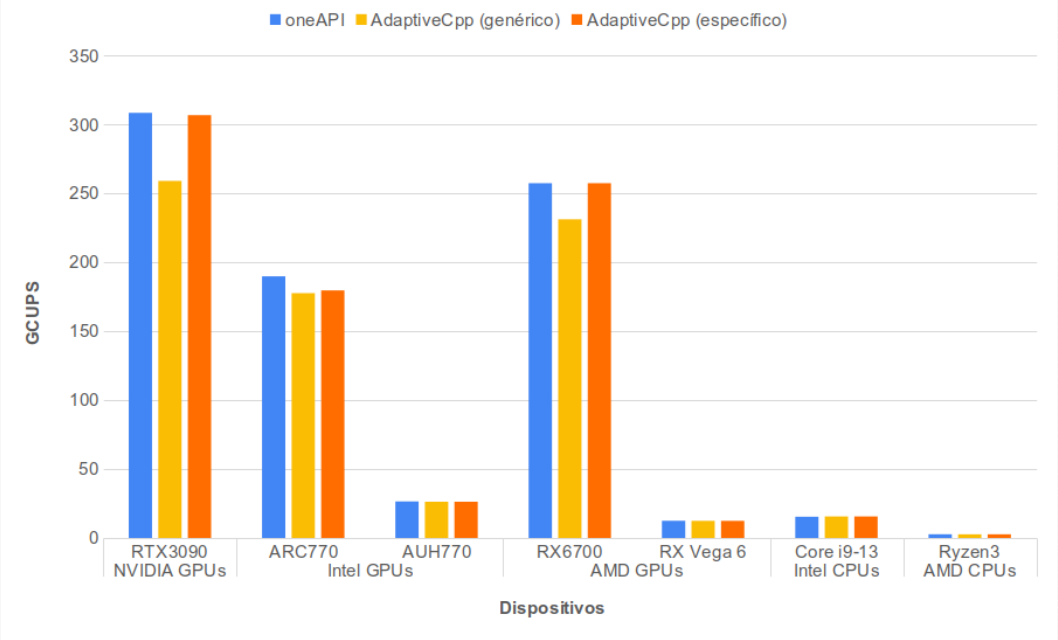

Figura 4.11: Comparación de rendimiento entre implementaciones SYCL (oneAPI y AdaptiveCpp) en diferentes GPUs y CPUs de distintos proveedores.

ferencias significativas de rendimiento entre oneAPI y AdaptiveCpp, excepto para la GPU Arc A770 donde el primero es solo 5 % más rápido que el segundo. Este hecho contribuye a la interoperabilidad de SYCL y facilita su adopción, dado que posibilita a los programadores emplear diversas implementaciones de SYCL sin preocuparse por la disminución del rendimiento. Esto resulta especialmente relevante para aquellos desarrolladores que trabajan en proyectos que demandan un alto rendimiento, ya que pueden seleccionar la implementación de SYCL que mejor se ajuste a sus requerimientos.

4.3. Trabajos relacionados y discusión

Algunos estudios preliminares han comparado el rendimiento entre SYCL y CUDA en diferentes dominios.

En [160], los autores utilizaron ADEPT como caso de estudio, el cual consiste de un *kernel* de alineamiento de lecturas cortas acelerado por GPU. Al utilizar una GPU NVIDIA A100, encontraron que la implementación de SYCL se ejecuta aproximadamente 2× más lento que su contraparte de CUDA en todos los experimentos. Los autores atribuyen esta discrepancia a la mayor utilización de la memoria caché por parte de CUDA y al mayor uso de registros por parte de SYCL. Además, los autores verificaron la portabilidad del código de SYCL en una iGPU Intel P630.

En [16], los autores profundizan en el proceso de migración de una aplicación CPU-GPU para la detección de epistasis de CUDA a SYCL, encontrando que el rendimiento más alto de ambas versiones es comparable en una GPU NVIDIA V100. Sin embargo, es importante destacar que se requirió de ajuste manual en la implementación de SYCL para alcanzar su rendimiento máximo. Al investigar el código PTX, los autores observaron que SYCL no realiza las mismas optimizaciones que CUDA, como el desenrollado de bucles.

En [159], los autores identificaron brechas de rendimiento en varias aplicaciones de bioinformática. El estudio involucró la selección de aplicaciones de código abierto que habían sido migradas de CUDA a SYCL, seguido de una evaluación exhaustiva de su rendimiento en una GPU NVIDIA V100. A través del análisis de perfilado, los autores encontraron que el compilador de SYCL carece de ciertas optimizaciones que la versión de CUDA sí tiene, incluyendo la gestión de memoria, la vectorización de instrucciones y el desenrollado de bucles, entre otros.

En [161], los autores comparan dos versiones de AutoDock-GPU, una en CUDA y otra en SYCL, en una CPU Intel Xeon Platinum 8360Y, una GPU NVIDIA A100 y una GPU Intel Max 1550. En la GPU A100, SYCL muestra un rendimiento más lento en algunos casos en comparación con CUDA, con ratios de rendimiento que van desde 1.24× hasta 2.38×. Sin embargo, en los casos de prueba pequeños, SYCL supera a CUDA en 1.09×. Los autores atribuyen los ratios más bajos al esfuerzo de sincronización requerido en regiones intensivas en cómputo, como la función de puntuación y el cálculo del gradiente. Destacan la necesidad de un análisis de rendimiento más profundo y sugieren una mayor optimización, especialmente en áreas intensivas en cómputo, para mejorar el rendimiento de SYCL.

En [170], los autores analizan el rendimiento de mini-aplicaciones creadas tanto en SYCL como en CUDA, ejecutándose en una GPU NVIDIA V100. Aunque hay algunas características que no son totalmente compatibles, el rendimiento de SYCL es comparable al de CUDA. Además, las diferencias de rendimiento se deben en gran medida a variaciones en los patrones de acceso a la memoria.

En [154], los autores evalúan la brecha entre el rendimiento y la portabilidad del código en aceleradores HPC utilizando el conocido algoritmo de *k-means*, comparando SYCL con CUDA y OpenMP en GPUs Intel Max 1100, NVIDIA A100 y AMD MI250X. La implementación de SYCL muestra un rendimiento superior en las GPU y CPU de Intel, y un rendimiento equivalente en las GPUs de NVIDIA, además de ofrecer una posible compatibilidad entre diferentes fabricantes.

En [171] el autor analiza la portabilidad y rendimiento de SYCL en GPU Maxs 1100 de Intel, A10 de NVIDIA y MI250X de AMD, además de CPUs Xeon Platinum 8360Y de Intel, EPYC 9V33X de AMD y Ampere Altra. En particular, se enfoca en aplicaciones con limitaciones de ancho de banda, utilizando dos DSLs (OPS y OP2) para diferentes paralelizaciones. Al comparar SYCL con enfoques de programación "nativos" (CUDA, HIP, OpenMP), el autor encuentra que el primero, en promedio, iguala o supera el rendimiento de los enfoques nativos en GPUs; por el contrario, resulta menos eficiente en CPUs. Por

último, destaca la necesidad de optimización adicional para mejorar el rendimiento de SYCL en CPUs y la importancia de elegir el enfoque de paralelización correcto para maximizar la eficiencia.

En el estudio realizado por los autores en [172], se pone énfasis en la implementación y optimización de CRK-HACC, una aplicación de cosmología a gran escala, utilizando SYCL en GPUs Intel Max 1550, NVIDIA A100 y AMD Instinct MI250X; así como CPUs de Intel Xeon Max 9470C, AMD EPYC 7543P y AMD EPYC 7A53. Se describe la migración del código original de CUDA a SYCL utilizando SYCLomatic, evidenciando que la especialización de los *kernels* para objetivos específicos mejora considerablemente la portabilidad del rendimiento sin afectar en gran medida la productividad del programador. La versión SYCL de CRK-HACC logra una portabilidad de rendimiento de 0.96 con una divergencia de código prácticamente nula, lo cual demuestra la viabilidad de SYCL para aplicaciones portables en términos de rendimiento.

En [173], el autor se enfoca en evaluar la portabilidad del rendimiento del método clásico de Gradiente Conjugado, en diferentes plataformas de hardware utilizando SYCL. Las plataformas incluyen una CPU Intel Xeon Gold 6128, una GPU NVIDIA Quadro GP100 y una FPGA Intel Arria 10 GX 1150. Se comparan las implementaciones nativas y basadas en SYCL en cada plataforma, analizando los tiempos de ejecución y la portabilidad del rendimiento. El estudio demuestra que SYCL logra una buena portabilidad de rendimiento en la CPU y GPU en comparación con las implementaciones de OpenMP y CUDA, respectivamente. Sin embargo, el rendimiento obtenido en las FPGAs no fue satisfactorio, lo que plantea la posibilidad de optimizar el código SYCL en este tipo de dispositivos en futuros trabajos de investigación.

En el artículo [174], se presenta un estudio acerca de la implementación de métodos iterativos por lotes en GPUs Intel utilizando SYCL, y se compara con implementaciones previas en CUDA. Los investigadores concluyen que, en general, la implementación de SYCL muestra un rendimiento inferior en comparación con las implementaciones previas en CUDA para GPUs NVIDIA. No obstante, se observó que para ciertos tamaños de matrices y bajo condiciones de optimización específicas, SYCL puede alcanzar o incluso superar el rendimiento de CUDA. Esto sugiere que, aunque SYCL es prometedor, su rendimiento puede depender significativamente de la arquitectura de la GPU y de las optimizaciones específicas aplicadas.

En [175], los autores llevaron a cabo una evaluación de SYCL en el contexto de cálculos de FTLE en sistemas GPU heterogéneos. Se tomaron dos escenarios del mundo real como casos de estudio, y se comparó el rendimiento y el esfuerzo de desarrollo de SYCL con las versiones equivalentes en CUDA y HIP. Para ello, se utilizaron dos CPUs Intel Xeon Platinum 8160, una GPU NVIDIA Tesla V100 y una AMD Vega 10 XT Radeon PRO WX 9100 GPU. Los resultados obtenidos revelaron que SYCL no genera un *overhead* significativo en el tiempo de ejecución de los *kernels* de GPU, y que el esfuerzo de desarrollo para el código del *host* es menor en SYCL en comparación con CUDA o HIP. Además, se destacó la capacidad de SYCL para trabajar de manera más eficiente con dispositivos GPU de diferentes proveedores simultáneamente.

En [176] se lleva a cabo un análisis sobre la portabilidad y rendimiento de los *kernels* SYCL en GPUs. Los investigadores realizaron la migración de un *kernel* de bioinformática de CUDA a HIP y SYCL con el objetivo de estudiar su rendimiento en las GPUs NVIDIA V100 y AMD MI210. Los resultados revelaron que el kernel CUDA es aproximadamente un 25 % más rápido que el de SYCL en la GPU de NVIDIA. Además, el estudio resalta las diferencias en la generación de instrucciones GPU y las optimizaciones de compiladores entre SYCL, CUDA y HIP. Como conclusión, se destaca que las optimizaciones en tamaños de direcciones de memoria, anchuras de accesos a memoria y accesos a subpalabras pueden

contribuir a reducir la brecha de rendimiento y mejorar la portabilidad de rendimiento de SYCL.

En [177], los autores analizan la portabilidad y el rendimiento de SYCL en la generación de números pseudoaleatorios. El estudio se enfoca en la aplicación de mapeo de conectividad de expresión génica, que originalmente se aceleraba con CUDA y la generación rápida de números pseudoaleatorios en GPU. Los autores describen la experiencia de migrar la generación de números pseudoaleatorios de CUDA a SYCL de manera manual y evalúan el rendimiento de los generadores de números pseudoaleatorios utilizando la librería de CUDA. Los resultados indican que el rendimiento de SYCL es comparable al de CUDA, pero se requiere un profundo entendimiento de la interfaz oneMKL y consideraciones adicionales en la gestión de la memoria y la optimización del rendimiento.

En [178], los autores abordaron la migración de la suite de *benchmarking Altis*, que originalmente estaba en CUDA, a SYCL para su uso en GPUs y FPGAs. Utilizaron la herramienta *dpct* para llevar a cabo la migración y optimización del código para diferentes plataformas. Los experimentos se ejecutaron en varios dispositivos, incluyendo GPUs NVIDIA (RTX 2080, A100) y FPGAs Intel (Stratix 10, Agilex). A partir de los resultados obtenidos, se observaron diferencias significativas de rendimiento entre las implementaciones en CUDA y SYCL, siendo SYCL la que obtuvo un rendimiento inferior, lo cual requirió de una optimización específica para cada dispositivo. Los autores se centraron en mejorar el rendimiento en FPGAs, aplicando técnicas como la replicación de unidades de cómputo y optimizaciones de tipo de datos. En cuanto a las GPUs, se realizaron ajustes en el desenrollado de bucles y en las opciones de compilación de Clang.

En [179], se lleva a cabo un análisis del rendimiento de GROMACS, un paquete de simulación de dinámica molecular, implementado tanto en SYCL como en CUDA. Se realizaron pruebas utilizando diferentes conjuntos de datos en GPUs NVIDIA (P100, V100, A100), comparando las versiones SYCL y CUDA de GROMACS. Los resultados indican que, aunque CUDA mostró un mejor rendimiento para el conjunto de datos *5NM_WATER*, SYCL superó a CUDA en los conjuntos de datos más grandes, como benchMEM, 10 nm y 15 nm. Este comportamiento posiblemente se debe a la capacidad de SYCL para aprovechar optimizaciones de hardware, como la librería Intel MKL y las instrucciones SIMD, para cálculos FFT acelerados. Los autores sugieren que SYCL puede ser una alternativa viable a CUDA para simulaciones de dinámica molecular, ofreciendo mayor flexibilidad y potencial para mejorar la portabilidad entre diferentes arquitecturas de computadoras sin comprometer el rendimiento.

Por último, algunos trabajos han comparado el rendimiento y la portabilidad de diferentes implementaciones de SYCL [180, 181, 182]. En general, los resultados de rendimiento fueron similares entre las diferentes implementaciones de SYCL, aunque se han producido diferencias significativas en algunos casos aislados. Sin embargo, debido a la rápida evolución de estas herramientas, no se pueden considerar como definitivos a sus resultados.

En este estudio, tal como se demostró a lo largo de los capítulos, SW# fue migrado completamente con una mínima intervención manual por parte del programador para analizar la portabilidad funcional y de rendimiento de SYCL. Es importante resaltar que este análisis abarcó un conjunto de plataformas más amplio y diverso que cualquier otro trabajo previo en este campo. Se logró verificar la portabilidad funcional del código migrado entre diferentes arquitecturas de GPU y CPU de múltiples proveedores, incluyendo 6 GPUs de NVIDIA con 5 microarquitecturas diferentes, 3 microarquitecturas de GPU de Intel (una discreta y dos integradas), 2 microarquitecturas de GPU de AMD (una discreta y una integrada), 1 microarquitectura de CPU de AMD y 7 microarquitecturas de CPU de Intel (de diferentes segmentos). Además, el análisis tuvo en cuenta diferentes variantes de la aplicación ASB, di-

versas implementaciones de SYCL y su comportamiento en configuraciones puras e híbridas de CPU y GPU. En cuanto a prestaciones, el código migrado no solo demostró ser portable en términos de funcionalidad, sino también en cuanto a rendimiento. En relación con esto, no se observó *overhead* en el rendimiento de las GPUs de NVIDIA para una amplia variedad de configuraciones de problema, mientras que la eficiencia arquitectónica se mantuvo al mismo nivel con las GPUs de otros fabricantes (como Intel y AMD). Más destacado aún es el hecho de que el código SYCL alcanzó el mismo nivel de eficiencia al ejecutarse en CPUs de diferentes proveedores. En la misma línea, también se puede remarcar que el rendimiento fue prácticamente el mismo al cambiar de compilador SYCL, independientemente del dispositivo CPU o GPU utilizado. Sin embargo, en configuraciones multi-GPU y CPU-GPU no se logró el mismo resultado. En estos casos, la eficiencia arquitectónica disminuyó debido a razones ajenas a SYCL, como una menor proporción de trabajo por recurso computacional y a una distribución de carga subóptima de SW#. Más allá de eso, es importante destacar que la portabilidad funcional en estas arquitecturas se logró sin ningún costo adicional más que volver a compilar el código (aunque esto puede mejorarse con los avances en los compiladores).

Los resultados demuestran que SYCL mantiene una portabilidad funcional y de rendimiento en diferentes GPUs y CPUs, lo que establece un precedente importante. Esto permite que la comunidad pueda portar sus aplicaciones a una amplia gama de plataformas, sin necesidad de reescribir o adaptar significativamente el código para cada una, lo que incrementa la productividad. Sin embargo, es importante reconocer que, como se ha observado en algunos trabajos relacionados, aún existen ciertas limitaciones en SYCL para el desarrollo de aplicaciones, generalmente vinculadas a cuestiones específicas. Afortunadamente, las implementaciones están evolucionando rápidamente para superar estas barreras y, al mismo tiempo, la comunidad de usuarios y desarrolladores de SYCL parece estar en constante crecimiento. De esta manera, SYCL se posiciona como un modelo de programación unificado, portable y eficiente para sistemas heterogéneos basados en GPUs en el ámbito de la bioinformática.

Capítulo 5

Conclusiones y Trabajos Futuros

En este apartado se presentan las conclusiones de este estudio (Sección 5.1) junto con las potenciales áreas de investigación futura (Sección 5.2).

5.1. Conclusiones

En la última década, la búsqueda por mejorar la eficiencia energética de los sistemas de cómputo ha impulsado la tendencia hacia la computación heterogénea y las arquitecturas masivamente paralelas. En la actualidad, las GPUs pueden considerarse los aceleradores dominantes, siendo NVIDIA, Intel y AMD los fabricantes más destacados. Esto plantea un desafío significativo para los investigadores que utilizan GPUs en sus experimentos y simulaciones. Una cuestión crítica es cómo aprovechar esta creciente capacidad computacional de manera transparente sin tener que prestar atención a los modelos de programación, el soporte de hardware o el ecosistema obligatorio de software.

Desde el punto de vista de la programación, CUDA sigue siendo el lenguaje de programación más popular para las GPUs, aunque al ser propietario, solo resulta válido para dispositivos NVIDIA. Afortunadamente, otras iniciativas abiertas han contemplado la programación de GPUs e incluso de otros aceleradores de forma genérica. En particular, SYCL surge recientemente como una opción especialmente prometedora para convertirse en un estándar unificado para la programación heterogénea paralela. Una característica destacada de SYCL es su condición de capa de abstracción multiplataforma, que permite a los programadores adherirse al principio fundamental de escribir código una vez y ejecutarlo en cualquier lugar". En este sentido, el mismo código SYCL puede ejecutarse no solo en múltiples GPUs de diferentes proveedores, sino también en diferentes plataformas de hardware, incluyendo CPUs y FPGAs. De esta manera, SYCL busca reducir los costos de desarrollo y mantenimiento, así como mejorar la productividad de la programación.

La bioinformática y la biología computacional son dos campos que han estado explotando el uso de GPUs durante más de dos décadas, y muchas de sus implementaciones se basan en CUDA, lo que impone limitaciones significativas en cuanto a la portabilidad en una amplia gama de arquitecturas heterogéneas. Por ese motivo, esta tesis se ha planteado como objetivo general *evaluar la viabilidad de SYCL como un modelo de programación heterogénea unificado, portable y eficiente para el diseño y desarrollo de aplicaciones con alta demanda computacional en sistemas heterogéneos basados en GPUs, específicamente en el ámbito de*

la bioinformática. A continuación, se exponen los objetivos específicos y se explica cómo se lograron:

- *Investigar y comparar críticamente los modelos de programación y métricas de rendimiento presentes en el contexto de la computación heterogénea, específicamente en aplicaciones bioinformáticas, con el propósito de establecer una base conceptual para esta investigación.*

En cumplimiento de este objetivo, se llevó a cabo una exhaustiva revisión bibliográfica en el Capítulo 2, abordando los modelos de programación heterogénea más relevantes, como CUDA, OpenCL, OpenMP, OpenACC, Kokkos, RAJA, y SYCL. También se investigaron métricas clave de evaluación, como el rendimiento, la portabilidad y el esfuerzo de programación. Además, se estudiaron conceptos fundamentales de bioinformática, centrándose en el ASB por representar una operación fundamental con amplias aplicaciones en diversas áreas de la biología y la medicina. Finalmente, se estudió la aceleración de ASB y se relevaron las implementaciones basadas en GPU existentes. Esta base conceptual estableció los fundamentos teóricos indispensables para los análisis experimentales posteriores.

- *Diseñar y desarrollar software que aproveche las capacidades de SYCL para sistemas heterogéneos basados en GPUs en el contexto de la bioinformática, considerando especialmente la migración de aplicaciones implementadas en CUDA.*

Con el fin de alcanzar este objetivo, se seleccionó SW# como caso de estudio debido a que es una suite para ASB basada en CUDA, que presenta un rendimiento destacado y provee amplia funcionalidad, como se mostró en el Capítulo 3. Luego, utilizando la herramienta SYCLomatic, se migró por completo el código CUDA de SW# a SYCL siguiendo un riguroso proceso que incluyó la ejecución de la herramienta, la modificación del código generado, la corrección de errores en tiempo de ejecución, la verificación funcional, las optimizaciones y la estandarización de código SYCL. Este proceso resultó en una versión de SW# implementada en SYCL, completamente funcional y apta para aprovechar eficientemente los sistemas heterogéneos basados en GPUs.

En conclusión, SYCLomatic es una solución útil para migrar código de CUDA a SYCL, ya que puede convertir una gran parte del código que sigue patrones comunes. Esto reduce errores y ahorra tiempo y esfuerzo en la programación. Sin embargo, es importante tener en cuenta que no es una solución completa y puede requerir intervención manual en casos donde no hay un equivalente directo en SYCL o se necesiten optimizaciones particulares. Resulta importante notar que aunque SYCLomatic es útil, no es indispensable para la migración de código CUDA a SYCL. Los desarrolladores pueden optar por una migración manual, lo que les permite explorar diferentes estrategias de optimización o elegir otras características de SYCL de las que la herramienta utiliza por defecto. Adicionalmente, también puede conducir a un código resultante más sintético y *elegante*, considerando que no será producto de una traducción automática. Sin embargo, se debe tener en cuenta que el esfuerzo de programación será significativamente mayor, especialmente en aplicaciones de gran cantidad de líneas de código. En estos escenarios, podría ser más productivo modificar el código ya migrado por SYCLomatic en lugar de comenzar el proceso de migración desde cero.

- *Medir y comparar las prestaciones del software desarrollado en distintos sistemas heterogéneos basados en GPUs, considerando la portabilidad, el rendimiento y la productividad como parámetros de interés.*

Para evaluar las prestaciones de la versión SYCL de SW#, se llevaron a cabo múltiples experimentos en el Capítulo 4, donde se evaluó su portabilidad funcional y de rendimiento. Con este propósito, se realizaron experimentos en diversos contextos, que abarcaron la búsqueda del WG óptimo, la variabilidad ante cambios en la carga de trabajo (longitud de secuencia de consulta y tamaño de la base de datos por el lado de proteínas; longitudes de las secuencias en el caso de ADN) y en parámetros específicos del problema (esquema de puntuación y algoritmo de alineamiento). Para las pruebas, se incluyeron GPUs de distintos tipos, modelos y fabricantes, como NVIDIA, Intel y AMD, con el fin de abarcar una amplia variedad de configuraciones de hardware y capacidades de procesamiento. En particular, se utilizaron 6 GPUs de NVIDIA con 5 microarquitecturas diferentes, 3 microarquitecturas de GPU de Intel (una discreta y dos integradas) y 2 microarquitecturas de GPU de AMD (una discreta y una integrada). Esto permitió evaluar tanto la portabilidad funcional como de rendimiento de la implementación SYCL en diferentes entornos de GPU y multi-GPU. En cuanto a las CPUs, se incluyeron de distintas generaciones y modelos, abarcando 7 microarquitecturas de Intel (de diferentes segmentos) y 1 de AMD. Esta diversidad de CPUs permitió evaluar cómo la implementación SYCL se desempeñaba en combinación con diferentes configuraciones de CPU y cómo afectaba al rendimiento general del sistema. Con el fin de evaluar las capacidades de SW# en un entorno híbrido, se llevaron a cabo experimentos que combinaron iGPUs y dGPUs con CPUs. Por último, se realizaron experimentos con diferentes implementaciones de SYCL, que incluyeron DPC++ y AdaptiveCPP, con el propósito de analizar su interoperabilidad y rendimiento.

Es importante resaltar que el análisis de portabilidad funcional y de rendimiento abarcó un conjunto de plataformas más amplio y diverso que cualquier otro trabajo previo en este campo. Se logró verificar la portabilidad funcional del código migrado entre diferentes arquitecturas de GPU y CPU de múltiples proveedores, segmentos y generaciones. Además, el análisis tuvo en cuenta diferentes variantes de la aplicación ASB, distintas implementaciones de SYCL y su comportamiento en configuraciones puras e híbridas de CPU y GPU. En cuanto a prestaciones, el código migrado no solo demostró ser portable en términos de funcionalidad, sino también en cuanto a rendimiento. En relación con esto, no se observó *overhead* en el rendimiento de las GPUs de NVIDIA para una amplia variedad de configuraciones de problema, mientras que la eficiencia arquitectónica se mantuvo al mismo nivel con las GPUs de otros fabricantes (como Intel y AMD). Más destacado aún es el hecho de que el código SYCL alcanzó el mismo nivel de eficiencia al ejecutarse en CPUs de diferentes proveedores. En la misma línea, también se puede remarcar que el rendimiento fue prácticamente el mismo al cambiar de compilador SYCL, independientemente del dispositivo CPU o GPU utilizado. Sin embargo, en configuraciones multi-GPU y CPU-GPU no se logró el mismo resultado. En estos casos, la eficiencia arquitectónica disminuyó debido a razones ajenas a SYCL, como una menor proporción de trabajo por recurso computacional y a una distribución de carga subóptima de SW#. Más allá de eso, es importante destacar que la portabilidad funcional en estas arquitecturas se logró sin ningún costo adicional más que volver a compilar el código.

Los resultados demuestran que SYCL mantiene una portabilidad funcional y de rendimiento en diferentes GPUs y CPUs, lo que establece un precedente importante. Esto permite que la comunidad pueda portar sus aplicaciones a una amplia gama de plataformas, sin necesidad de reescribir o adaptar significativamente el código para cada una, lo que incrementa la productividad. Sin embargo, es importante reconocer que, como se ha observado en algunos trabajos relacionados, aún existen ciertas limitaciones en SYCL para el desarrollo de aplicaciones, generalmente vinculadas a cuestiones específicas. Afortunadamente, las implementaciones están evolucionando rápidamente para superar estas barreras y, al mismo

tiempo, la comunidad de usuarios y desarrolladores de SYCL parece estar en constante crecimiento. De esta manera, SYCL se posiciona como un modelo de programación unificado, portable y eficiente para sistemas heterogéneos basados en GPUs en el ámbito de la bioinformática.

De acuerdo a los resultados obtenidos y a las contribuciones realizadas, se espera que esta tesis contribuya al avance de la programación unificada para sistemas heterogéneos basados en GPUs en bioinformática, abriendo nuevas oportunidades para el desarrollo de aplicaciones eficientes y portables en este campo.

5.2. Trabajos futuros

Algunas líneas de trabajo a futuro pueden desprenderse de esta tesis:

- Optimizar el código SYCL para alcanzar su máximo rendimiento. En particular, la suite original SW# no considera algunas optimizaciones conocidas para el alineamiento SW [135], como la reordenación de instrucciones para reducir la cantidad de las mismas y el uso de enteros de menor precisión para aumentar la paralelización [1]. Además, se busca mejorar la estrategia de distribución de carga de trabajo al utilizar más de un dispositivo. Estas mejoras conducirán a tasas de eficiencia más altas.
- Ejecutar el código SYCL en otras arquitecturas FPGAs y considerar otros modelos de programación como Kokkos y RAJA, para fortalecer el estudio actual de portabilidad de rendimiento.

[1] Es importante destacar que en el momento del desarrollo de SW#, la mayoría de las GPUs habilitadas para CUDA no admitían operaciones aritméticas eficientes en tipos de datos vectoriales de 8 bits.

Bibliografía

[1] H. Giefers et al. «Analyzing the energy-efficiency of sparse matrix multiplication on heterogeneous systems: A comparative study of GPU, Xeon Phi and FPGA». En: *2016 IEEE ISPASS*. 2016, págs. 46-56.

[2] Mohamed Zahran. «Heterogeneous computing: Here to stay». En: *Communications of the ACM* 60.3 (2017), págs. 42-45.

[3] William J. Dally, Yatish Turakhia y Song Han. «Domain-Specific Hardware Accelerators». En: *Commun. ACM* 63.7 (jun. de 2020), págs. 48-57. ISSN: 0001-0782. DOI: 10.1145/3361682. URL: https://doi.org/10.1145/3361682.

[4] M. Snir. «The future of supercomputing». En: *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. 2018, págs. 172-172.

[5] Daniel Reed, Dennis Gannon y Jack Dongarra. *Reinventing High Performance Computing: Challenges and Opportunities*. Mar. de 2022.

[6] Enzo Rucci. «Evaluación de rendimiento y eficiencia energética de sistemas heterogéneos para bioinformática». En: *Tesis Doctoral, UNLP* (2016).

[7] Marco S Nobile et al. «Graphics processing units in bioinformatics, computational biology and systems biology». En: *Briefings in Bioinformatics* 18.5 (jul. de 2016), págs. 870-885. ISSN: 1467-5463. DOI: 10.1093/bib/bbw058.

[8] João Carrasqueira. *AMD gains market share as GPU shipments increase in Q2 2023*. Sep. de 2023. URL: https://www.neowin.net/news/amd-gains-market-share-as-gpu-shipments-increase-in-q2-2023/ (visitado 26-12-2023).

[9] David B. Kirk y Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. ISBN: 0123814723.

[10] Robert Dow. *GPU shipments increase year-over-year in Q3*. https://www.jonpeddie.com/press-releases/gpu-shipments-increase-year-over-year-in-q3. 2021.

[11] J. E. Stone, D. Gohara y Guochun Shi. «OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems». En: *Computing in Science and Engg.* 12.3 (mayo de 2010), págs. 66-73. ISSN: 1521-9615.

[12] *Programming/OpenCL*. URL: https://hpc.mediawiki.hull.ac.uk/Programming/OpenCL (visitado 26-12-2023).

[13] *The OpenMP Specification*. http://https://www.openmp.org/.

[14] *The OpenACC Specification*. http://https://www.openacc.org/.

[15] Rob Farber. *Parallel Programming with OpenACC*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016. ISBN: 0124103979.

[16] Zheming Jin y Jeffrey S. Vetter. «Performance Portability Study of Epistasis Detection Using SYCL on NVIDIA GPU». En: *Proceedings of the 13th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*. BCB '22. Northbrook, Illinois: Association for Computing Machinery, 2022. ISBN: 9781450393867. DOI: 10.1145/3535508.3545591. URL: https://doi.org/10.1145/3535508.3545591.

[17] Khronos SYCL working group. *SYCL 2020 specification*. https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html. 2023.

[18] Ronan Keryell y Lin-Ya Yu. «Early Experiments Using SYCL Single-Source Modern C++ on Xilinx FPGA». En: *Proceedings of the IWOCL '18*. Oxford, UK: ACM, 2018. DOI: 10.1145/3204919.3204937.

[19] Codeplay Software. *ComputeCpp Comunity Edition*. https://developer.codeplay.com/products/computecpp/ce/home. 2023.

[20] Ben Ashbaugh et al. «Data Parallel C++ Enhancing SYCL Through Extensions for Productivity and Performance». En: *Proceedings of the International Workshop on OpenCL*. 2020, págs. 1-2.

[21] *The triSYCL project*. https://github.com/triSYCL/triSYCL. 2023.

[22] Aksel Alpay. *OpenSYCL implementation*. https://github.com/OpenSYCL/OpenSYCL. 2023.

[23] Nikita Hariharan et al. «Heterogeneous Programming using OneAPI». En: *Parallel Universe* 39 (2020), págs. 5-18.

[24] Intel. *SYCLomatic: A New CUDA*-to-SYCL*Code Migration Tool*. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/syclomatic-new-cuda-to-sycl-code-migration-tool.html (visitado 26-12-2023).

[25] T. K. Atwood y D. J. Parry-Smith. «Introducción a la bioinformática». En: *Prentice Hall* (2012).

[26] Edans F. De Oilveira Sandes, Azzedine Boukerche y Alba C. M. A. De Melo. «Parallel Optimal Pairwise Biological Sequence Comparison: Algorithms, Platforms, and Classification». En: *ACM Comput. Surv.* 48.4 (2016). DOI: 10.1145/2893488.

[27] Styliani Loukatou et al. «Molecular dynamics simulations through GPU video games technologies». En: *Journal of molecular biochemistry* 3.2 (2014), pág. 64.

[28] Masahito Ohue et al. «MEGADOCK 4.0: an ultra–high-performance protein–protein docking software for heterogeneous supercomputers». En: *Bioinformatics* 30.22 (2014), págs. 3281-3283.

[29] Dariusz Mrozek, Miłosz Brożek y Bożena Małysiak-Mrozek. «Parallel implementation of 3D protein structure similarity searches using a GPU and the CUDA». En: *Journal of molecular modeling* 20.2 (2014), págs. 1-17.

[30] Jie Bao et al. «Efficient Implementation of MrBayes on Multi-GPU». En: *Molecular biology and evolution* 30 (mar. de 2013). DOI: 10.1093/molbev/mst043.

[31] Edans Flavius O. Sandes y Alba Cristina M.A. de Melo. «CUDAlign: Using GPU to Accelerate the Comparison of Megabase Genomic Sequences». En: *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '10. Bangalore, India: Association for Computing Machinery, 2010, págs. 137-146. ISBN: 9781605588773. DOI: 10.1145/1693453.1693473. URL: https://doi.org/10.1145/1693453.1693473.

[32] Edans Sandes y Alba Melo. «Smith-Waterman Alignment of Huge Sequences with GPU in Linear Space». En: jun. de 2011, págs. 1199-1211. DOI: 10.1109/IPDPS.2011.114.

[33] Edans Sandes y Alba Melo. «Retrieving Smith-Waterman Alignments with Optimizations for Megabase Biological Sequences Using GPU». En: *Parallel and Distributed Systems, IEEE Transactions on* 24 (mayo de 2013), págs. 1009-1021. DOI: 10.1109/TPDS.2012.194.

[34] Edans Sandes et al. «CUDAlign 3.0: Parallel Biological Sequence Comparison in Large GPU Clusters». En: mayo de 2014, págs. 160-169. ISBN: 978-1-4799-2784-5. DOI: 10.1109/CCGrid.2014.18.

[35] Edans Flavius de Oliveira Sandes et al. «CUDAlign 4.0: Incremental Speculative Traceback for Exact Chromosome-Wide Alignment in GPU Clusters». En: *IEEE Transactions on Parallel and Distributed Systems* 27.10 (2016), págs. 2838-2850. DOI: 10.1109/TPDS.2016.2515597.

[36] Matija Korpar y Mile Sikic. «SW# - GPU-enabled exact alignments on genome scale.» En: *Bioinformatics* 29.19 (2013), págs. 2494-2495. DOI: 10.1093/bioinformatics/btt410.

[37] Matija Korpar et al. «SWdb: GPU-Accelerated Exact Sequence Similarity Database Search». En: *PLOS ONE* 10.12 (dic. de 2016), págs. 1-11. DOI: 10.1371/journal.pone.0145857.

[38] Yongchao Liu, Douglas L Maskell y Bertil Schmidt. «CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units». en. En: *BMC Res. Notes* 2.1 (mayo de 2009), pág. 73.

[39] Yongchao Liu, Bertil Schmidt y Douglas L Maskell. «CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions». en. En: *BMC Res. Notes* 3.1 (abr. de 2010), pág. 93.

[40] Yongchao Liu, Adrianto Wirawan y Bertil Schmidt. «CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions». En: *BMC Bioinformatics* 14 (2013), págs. 117-117. URL: https://api.semanticscholar.org/CorpusID:4889656.

[41] Bertil Schmidt et al. «CUDASW++4.0: Ultra-fast GPU-based Smith-Waterman Protein Sequence Database Search». En: (oct. de 2023). DOI: 10.1101/2023.10.09.561526.

[42] Muaaz Gul Awan et al. «ADEPT: a domain independent sequence alignment strategy for gpu architectures». En: *BMC bioinformatics* 21 (sep. de 2020), pág. 406. DOI: 10.1186/s12859-020-03720-1.

[43] John L. Hennessy y David A. Patterson. «A New Golden Age for Computer Architecture». En: *Commun. ACM* 62.2 (ene. de 2019), págs. 48-60. ISSN: 0001-0782. DOI: 10.1145/3282307. URL: https://doi.org/10.1145/3282307.

[44] Shahla Gul y Noman Aftab. «A Comparison between RISC and CISC Microprocessor Architectures». En: *IJSEAT* 4.5 (2016). ISSN: 2321-6905. URL: http://www.ijseat.com/index.php/ijseat/article/view/663.

[45] Hiroshi Iwai. «End of the scaling theory and Moore's law». En: *2016 16th International Workshop on Junction Technology (IWJT)*. 2016, págs. 1-4. DOI: 10.1109/IWJT.2016.7486661.

[46] Erlin Yao et al. «Extending Amdahl's Law in the Multicore Era». En: *SIGMETRICS Performance Evaluation Review* 37 (oct. de 2009), págs. 24-26. DOI: 10.1145/1639562.1639571.

[47] R. Cypher y J. Sanz. «The SIMD Model of Parallel Computation». En: (1998), págs. I-V, 1-149. DOI: 10.1007/978-1-4612-2612-3.

[48] J. C. Mejía y M. O'Keefe. «High performance instruction memory design for multiprocessors». En: *[1993] Proceedings of the Twenty-sixth Hawaii International Conference on System Sciences* i (1993), 224-231 vol.1. DOI: 10.1109/HICSS.1993.270742.

[49] Anish Krishnakumar et al. «Domain-Specific Architectures: Research Problems and Promising Approaches». En: *ACM Trans. Embed. Comput. Syst.* 22.2 (ene. de 2023). ISSN: 1539-9087. DOI: 10.1145/3563946. URL: https://doi.org/10.1145/3563946.

[50] *High-Performance Computing Trends*. Springer, Cham, 2019, págs. 269-273. DOI: 10.1007/978-3-030-18338-7_16.

[51] W. Dally, S. Keckler y D. Kirk. «Evolution of the Graphics Processing Unit (GPU)». En: *IEEE Micro* 41 (2021), págs. 42-51. DOI: 10.1109/mm.2021.3113475.

[52] James Barker y Josh Bowden. «Manycore Parallelism through OpenMP - High-Performance Scientific Computing with Xeon Phi». En: (2013), págs. 45-57. DOI: 10.1007/978-3-642-40698-0_4.

[53] Mário Véstias y Horácio Neto. «Trends of CPU, GPU and FPGA for high-performance computing». En: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014, págs. 1-6. DOI: 10.1109/FPL.2014.6927483.

[54] E. Sanchez. «Field Programmable Gate Array (FPGA) Circuits». En: (1995), págs. 1-18. DOI: 10.1007/3-540-61093-6_1.

[55] Amna Shahid y Malaika Mushtaq. «A Survey Comparing Specialized Hardware And Evolution In TPUs For Neural Networks». En: *2020 IEEE 23rd International Multitopic Conference (INMIC)*. 2020, págs. 1-6. DOI: 10.1109/INMIC50486.2020.9318136.

[56] Oliver Jakob Arndt et al. «Portable implementation of advanced driver-assistance algorithms on heterogeneous architectures». En: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Orlando / Buena Vista, FL, USA: IEEE, mayo de 2017.

[57] Yushuqing Zhang, Kai Lu y Wenzhe Zhang. «CLMalloc: contiguous memory management mechanism for large-scale CPU-accelerator hybrid architectures». En: *Third International Symposium on Computer Engineering and Intelligent Communications (ISCEIC 2022)*. Ed. por Xuexia Ye y Xianye Ben. Xi'an, China: SPIE, feb. de 2023.

[58] *Top500*. https://www.top500.org/.

[59] D. Luebke et al. «GPGPU: general-purpose computation on graphics hardware». En: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (2006). DOI: 10.1145/1188455.1188672.

[60] Stephen W. Keckler. «GPU computing and the road to extreme-scale parallel systems». En: *2011 IEEE International Symposium on Workload Characterization (IISWC)*. 2011, págs. 1-1. DOI: 10.1109/IISWC.2011.6114191.

[61] Mark J. Harris. «Many-core GPU computing with NVIDIA CUDA». En: (2008), pág. 1. DOI: 10.1145/1375527.1375528.

[62] M. Garland. «Parallel computing with CUDA». En: *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)* (2010), págs. 1-1. DOI: 10.1109/IPDPS.2010.5470378.

[63] M. Ujaldón y Ümit V. Çatalyürek. «High-performance signal processing on emerging many-core architectures using cuda». En: *2009 IEEE International Conference on Multimedia and Expo* (2009), págs. 1825-1828. DOI: 10.1109/ICME.2009.5202878.

[64] Ogier Maitre. «Understanding NVIDIA GPGPU Hardware». En: (2013), págs. 15-34. DOI: 10.1007/978-3-642-37959-8_2.

[65] Erik Lindholm et al. «NVIDIA Tesla: A Unified Graphics and Computing Architecture». En: *IEEE Micro* 28.2 (2008), págs. 39-55. DOI: 10.1109/MM.2008.31.

[66] NVIDIA Corporation. *NVIDIA's Next Generation CUDA TM Compute Architecture: Fermi TM*. 2023. URL: https://www.nvidia.com/content/pdf/fermi%5C%5Fwhite%5C%5Fpapers/nvidia%5C%5Ffermi%5C%5Fcompute%5C%5Farchitecture%5C%5Fwhitepaper.pdf (visitado 26-12-2023).

[67] Yuanzhe Li et al. «Improving performance of GPU code using novel features of the NVIDIA kepler architecture». en. En: *Concurr. Comput.* 28.13 (sep. de 2016), págs. 3586-3605.

[68] NVIDIA Corporation. *NVIDIA ADA GPU Architecture*. 2023. URL: https://images.nvidia.com/aem-dam/Solutions/Data-Center/l4/nvidia-ada-gpu-architecture-whitepaper-v2.1.pdf (visitado 06-12-2023).

[69] Itigic. *History of AMD: Its Most Outstanding Graphics Cards*. 2023. URL: https://itigic.com/es/history-of-amd-its-most-outstanding-graphics-cards/ (visitado 26-12-2023).

[70] Mike Mantor. «AMD Radeon™ HD 7970 with graphics core next (GCN) architecture». En: *2012 IEEE Hot Chips 24 Symposium (HCS)*. 2012, págs. 1-35. DOI: 10.1109/HOTCHIPS.2012.7476485.

[71] *AMD Launches 2nd-Gen Embedded R-Series APUs and CPUs*. Mayo de 2014. URL: https://www.datacenterknowledge.com/archives/2014/05/22/amd-launches-2nd-gen-embedded-r-series-apus-cpus.

[72] *AMD presenta Radeon RX Vega: gran potencia gráfica a precio contenido*. Dic. de 2017. URL: https://www.muycomputer.com/2017/07/31/radeon-rx-vega-precio/.

[73] Nathan Otterness y James H. Anderson. «AMD GPUs as an Alternative to NVIDIA for Supporting Real-Time Workloads». En: *Euromicro Conference on Real-Time Systems*. 2020. URL: https://api.semanticscholar.org/CorpusID:220272915.

[74] Yuhui Bao et al. «NaviSim: A Highly Accurate GPU Simulator for AMD RDNA GPUs». En: oct. de 2022. DOI: 10.1145/3559009.3569666.

[75] PcComponentes. *AMD RDNA 2: Información y Características*. 2023. URL: https://www.pccomponentes.com/amd-rdna-2-informacion-caracteristicas (visitado 26-12-2023).

[76] AMD. *AMD RDNA™ Architecture*. 2023. URL: https://www.amd.com/en/technologies/rdna (visitado 26-12-2023).

[77] *AMD Radeon RX 5700 Review*. https://www.techpowerup.com/review/amd-radeon-rx-5700/2.html.

[78] HP. *Tarjeta gráfica integrada vs dedicada: ¿Cuál es la mejor opción?* URL: https://www.hp.com/mx-es/shop/tech-takes/tarjeta-grafica-integrada-dedicada (visitado 06-12-2023).

[79] MuyComputer. *Todo sobre las GPUs integradas de Intel*. 2016. URL: https://www.muycomputer.com/2016/05/25/todo-sobre-gpus-integradas-de-intel/ (visitado 06-12-2023).

[80] TechSpot. *The Last Time Intel Tried to Make a Graphics Card*. 2023. URL: https://www.techspot.com/article/2125-intel-last-graphics-card/ (visitado 26-12-2023).

[81] Jarred Walton. *Intel Arc Alchemist: Release Date, Specs, Everything We Know*. Sep. de 2022. URL: https://www.tomshardware.com/news/intel-arc-alchemist-release-date-specs-pricing-all-we-know (visitado 26-12-2023).

[82] *Microarquitectura de los procesadores Intel Skylake detallada*. https://www.hdtecnologia.com/microarquitectura-de-los-procesadores-intel-skylake-detallada/.

[83] *Intel Arc A770 Review*. https://www.profesionalreview.com/2022/10/28/intel-arc-a770-review/.

[84] NVIDIA. *CUDA C Programming Guide*. Ago. de 2015. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visitado 26-12-2023).

[85] Jason Sanders y Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1st. Addison-Wesley Professional, 2010. ISBN: 0131387685.

[86] Raymond Tay. *OpenCL Parallel Programming Development Cookbook*. Packt Publishing, 2013. ISBN: 1849694524.

[87] *OpenMP: Advanced Task-Based, Device and Compiler Programming: 19th International Workshop on OpenMP, IWOMP 2023, Bristol, UK, September 13–15, 2023, Proceedings*. Bristol, United Kingdom: Springer-Verlag, 2023. ISBN: 978-3-031-40743-7.

[88] Sunita Chandrasekaran y Guido Juckeland. *OpenACC for Programmers: Concepts and Strategies*. 1st. Addison-Wesley Professional, 2017. ISBN: 0134694287.

[89] Christian R. Trott et al. «Kokkos 3: Programming Model Extensions for the Exascale Era». En: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), págs. 805-817. DOI: 10.1109/TPDS.2021.3097283.

[90] David A. Beckingsale et al. «RAJA: Portable Performance for Large-Scale Scientific Applications». En: *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 2019, págs. 71-81. DOI: 10.1109/P3HPC49587.2019.00012.

[91] David Beckingsale et al. «Performance Portable C++ Programming with RAJA». En: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. PPoPP '19. Washington, District of Columbia: Association for Computing Machinery, 2019, págs. 455-456. ISBN: 9781450362252. DOI: 10.1145/3293883.3302577. URL: https://doi.org/10.1145/3293883.3302577.

[92] James Reinders et al. *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. Springer Nature, 2021.

[93] Peter Thoman, Facundo Heredia y Thomas Fahringer. «On the Compilation Performance of Current SYCL Implementations». En: mayo de 2022, págs. 1-12. DOI: 10.1145/3529538.3529548.

[94] Intel Corp. *Intel oneAPI*. https://software.intel.com/en-us/oneapi. 2021.

[95] Aksel Alpay. *OpenSYCL implementation*. https://github.com/AdaptiveCpp/AdaptiveCpp. 2023.

[96] Aksel Alpay et al. «Exploring the Possibility of a HipSYCL-Based Implementation of OneAPI». En: *International Workshop on OpenCL*. IWOCL'22. Bristol, United Kingdom, United Kingdom: Association for Computing Machinery, 2022. ISBN: 9781450396585. DOI: 10.1145/3529538.3530005. URL: https://doi.org/10.1145/3529538.3530005.

[97] Aksel Alpay y Vincent Heuveline. «One Pass to Bind Them: The First Single-Pass SYCL Compiler with Unified Code Representation Across Backends». En: *Proceedings of the 2023 International Workshop on OpenCL*. IWOCL '23. Cambridge, United Kingdom: Association for Computing Machinery, 2023. DOI: 10.1145/3585341.3585351. URL: https://doi.org/10.1145/3585341.3585351.

[98] Yinan Ke, Mulya Agung e Hiroyuki Takizawa. «neoSYCL: a SYCL implementation for SX-Aurora TSUBASA». En: ene. de 2021, págs. 50-57. DOI: 10.1145/3432261.3432268.

[99] Peter Thoman, Daniel Gogl y Thomas Fahringer. «Sylkan: Towards a Vulkan Compute Target Platform for SYCL». En: *International Workshop on OpenCL*. IWOCL'21. Munich, Germany: Association for Computing Machinery, 2021. ISBN: 9781450390330. DOI: 10.1145/3456669.3456683. URL: https://doi.org/10.1145/3456669.3456683.

[100] *SYCL*. https://www.khronos.org/sycl/.

[101] Ami Marowka. «Toward a Better Performance Portability Metric». En: *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 2021, págs. 181-184. DOI: 10.1109/PDP52278.2021.00036.

[102] Ami Marowka. «On the Performance Portability of OpenACC, OpenMP, Kokkos and RAJA». En: HPCAsia2022. Virtual Event, Japan: Association for Computing Machinery, 2022, págs. 103-114. ISBN: 9781450384988. DOI: 10.1145/3492805.3492806. URL: https://doi.org/10.1145/3492805.3492806.

[103] S.J. Pennycook, J.D. Sewall y V.W. Lee. «Implications of a metric for performance portability». En: *Future Generation Computer Systems* 92 (2019), págs. 947-958. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2017.08.007.

[104] Ami Marowka. «Reformulation of the performance portability metric». En: *Software: Practice and Experience* 52.1 (2022), págs. 154-171. DOI: https://doi.org/10.1002/spe.3002.

[105] Kaushal Bhatt, Vinit Tarey y Pushpraj Patel. «Analysis Of Source Lines Of Code(SLOC) Metric». En: *IJETAE* 2 (abr. de 2012).

[106] Sonam Bhatia y Jyoteesh Malhotra. «A survey on impact of lines of code on software complexity». En: *2014 International Conference on Advances in Engineering & Technology Research (ICAETR - 2014)*. 2014, págs. 1-4. DOI: 10.1109/ICAETR.2014.7012875.

[107] Jeff Gauthier et al. «A brief history of bioinformatics». En: *Briefings in Bioinformatics* 20.6 (ago. de 2018), págs. 1981-1996. ISSN: 1477-4054. DOI: 10.1093/bib/bby063. URL: https://doi.org/10.1093/bib/bby063.

[108] Wei Zhang y Tianwen Wang. «Bioinformatics-aided protein sequence analysis and engineering». en. En: *Curr. Protein Pept. Sci.* 24.6 (2023), págs. 477-487.

[109] Robert E Farrell Jr. «Transcriptomes and bioinformatics». En: *RNA Methodologies*. Elsevier, 2023, págs. 679-696.

[110] «Bioinformatics in proteomics». en. En: *Curr. Pharm. Biotechnol.* 5.1 (feb. de 2004), págs. 79-88.

[111] Rhys Newell. «Bioinformatic methods for genome-centric metagenomics». Tesis doct. 2023.

[112] Pinkal H Patel, Adarsh Jha y G S Chakraborthy. «Role of bioinformatics in drug design and discovery». En: *Interdisciplinary Biotechnological Advances*. Singapore: Springer Nature Singapore, 2023, págs. 1-33.

[113] Sonali Patil y Annika Durve Gupta. «Bioinformatics and its application in computing biological data». En: *Information Retrieval in Bioinformatics*. Singapore: Springer Nature Singapore, 2022, págs. 133-154.

[114] Deidre Margareta et al. «The Role of Bioinformatics in Personalized Medicine». En: *Your Future Medical Treatment* 46 (2019), págs. 785-788.

[115] J. Felsenstein. *Inferring phylogenies*. Sinauer Associates, 2003.

[116] M P Miller et al. «Quantifying the intragenic distribution of human disease mutations». en. En: *Ann. Hum. Genet.* 67.6 (nov. de 2003), págs. 567-579.

[117] J W Thomas et al. «Comparative analyses of multi-species sequences from targeted genomic regions». en. En: *Nature* 424.6950 (ago. de 2003), págs. 788-793.

[118] Ewen F Kirkness et al. «The dog genome: survey sequencing and comparative analysis». en. En: *Science* 301.5641 (sep. de 2003), págs. 1898-1903.

[119] Barry G Hall. «Simple and accurate estimation of ancestral protein sequences». en. En: *Proc. Natl. Acad. Sci. U. S. A.* 103.14 (abr. de 2006), págs. 5431-5436.

[120] Bertil Schmidt. «Bioinformatics: High Performance Parallel Computer Architectures». En: *Bioinformatics: High Performance Parallel Computer Architectures* (jul. de 2010). DOI: 10.1201/EBK1439814888.

[121] Pablo Mier, Miguel A Andrade-Navarro y Antonio J Pérez-Pulido. «OrthoFind facilitates the discovery of homologous and orthologous proteins». en. En: *PLoS One* 10.12 (dic. de 2015), e0143906.

[122] Steve Minchin y Julia Lodge. «Understanding biochemistry: structure and function of nucleic acids». en. En: *Essays Biochem.* 63.4 (oct. de 2019), págs. 433-456.

[123] Ratul Chowdhury et al. «IPRO+/-: Computational Protein Design Tool Allowing for Insertions and Deletions». En: *Structure* 28.12 (2020), 1344-1357.e4. ISSN: 0969-2126. DOI: https://doi.org/10.1016/j.str.2020.08.003. URL: https://www.sciencedirect.com/science/article/pii/S0969212620302859.

[124] Rakesh Trivedi y Hampapathalu Adimurthy Nagarajaram. «Substitution scoring matrices for proteins - An overview». en. En: *Protein Sci.* 29.11 (nov. de 2020), págs. 2150-2163.

[125] Temple F. Smith y Michael S. Waterman. «Identification of Common Molecular Subsequences». En: *Journal of Molecular Biology* 147.1 (mar. de 1981), págs. 195-197.

[126] O. Gotoh. «An Improved Algorithm for Matching Biological Sequences». En: *Journal of Molecular Biology*. Vol. 162. Cargèse, France., 1981, págs. 705-708.

[127] Alexander Isaev y Michael Deem. «Introduction to Mathematical Methods in Bioinformatics». En: *Physics Today* 58 (oct. de 2005), págs. 83-. DOI: 10.1063/1.2138428.

[128] «Semi Global Pairwise Sequence Alignment Using New Chromosome Structure Genetic Algorithm». En: *Ingénierie Des Systèmes D'information* 27.1 (2022), págs. 67-74. DOI: 10.18280/isi.270108.

[129] Stefaan Van Ryssen. «An Introduction to Bioinformatics Algorithms by Neil C. Jones and Pavel A. Pevzner. MIT Press, Cambridge, MA, USA, 2004. 434 pp., illus. Trade. ISBN: 0-262-10106-8 ldots». En: *Leonardo* 39 (oct. de 2006). DOI: 10.1162/leon.2006.39.5.486a.

[130] *International Nucleotide Sequence Database Collaboration*. URL: http://www.insdc.org/ (visitado 26-12-2023).

[131] *National Center for Biotechnology Information*. URL: http://www.ncbi.nlm.nih.gov/ (visitado 26-12-2023).

[132] *DNA Data Bank of Japan*. URL: http://www.ddbj.nig.ac.jp/ (visitado 26-12-2023).

[133] *European Molecular Biology Laboratory*. URL: http://www.embl.org/ (visitado 26-12-2023).

[134] M A Kentie. *Biological Sequence Alignment Using Graphics Processing Units*. 2010.

[135] T Rognes. «Faster Smith-Waterman database searches with inter-sequence SIMD parallelization». En: *BMC Bioinformatics* 12:221 (2011).

[136] Weiguo Liu et al. «Bio-sequence database scanning on a GPU». En: *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. 2006. DOI: 10.1109/IPDPS.2006.1639531.

[137] Yang Liu et al. «GPU Accelerated Smith-Waterman». En: *Computational Science – ICCS 2006*. Ed. por Vassil N. Alexandrov et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, págs. 188-195. ISBN: 978-3-540-34386-8.

[138] Svetlin A Manavski y Giorgio Valle. «CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment». en. En: *BMC Bioinformatics* 9 Suppl 2.S2 (mar. de 2008), S10.

[139] Dan Zou, Yong Dou y Fei Xia. «Optimization schemes and performance evaluation of Smith–Waterman algorithm on CPU, GPU and FPGA». En: *Concurrency and Computation: Practice & Experience* 24 (sep. de 2012), págs. 1625-1644. DOI: 10.1002/cpe.1913.

[140] Ayman Khalafallah et al. «Optimizing Smith-Waterman algorithm on Graphics Processing Unit». En: *2010 2nd International Conference on Computer Technology and Development*. 2010, págs. 650-654. DOI: 10.1109/ICCTD.2010.5645976.

[141] Plamenka Borovska y Milena Lazarova. «Parallel Models for Sequence Alignment on CPU and GPU». En: *Proceedings of the 12th International Conference on Computer Systems and Technologies*. CompSysTech '11. Vienna, Austria: Association for Computing Machinery, 2011, págs. 210-215. ISBN: 9781450309172. DOI: 10.1145/2023607.2023644. URL: https://doi.org/10.1145/2023607.2023644.

[142] Haidong Lan et al. «SWhybrid: A Hybrid-Parallel Framework for Large-Scale Protein Sequence Database Search». En: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2017, págs. 42-51. DOI: 10.1109/IPDPS.2017.42.

[143] Sven Warris et al. «pyPaSWAS: Python-based multi-core CPU and GPU sequence alignment». en. En: *PLoS One* 13.1 (ene. de 2018), e0190279.

[144] Edans F. De O. Sandes et al. «MASA: A Multiplatform Architecture for Sequence Aligners with Block Pruning». En: *ACM Trans. Parallel Comput.* 2.4 (feb. de 2016). ISSN: 2329-4949. DOI: 10.1145/2858656. URL: https://doi.org/10.1145/2858656.

[145] Huihui Zou et al. «ASW: Accelerating Smith–Waterman Algorithm on Coupled CPU–GPU Architecture». En: *International Journal of Parallel Programming* 47 (jun. de 2019). DOI: 10.1007/s10766-018-0617-3.

[146] Katherine Yelick, Leonid Oliker y Muaaz Gul Awan. *A Domain independent sequence alignment strategy for gpu architectures (ADEPT) v1.0*. 2019.

[147] S F Altschul et al. «Basic local alignment search tool». en. En: 215.3 (oct. de 1990), págs. 403-410.

[148] Intel Corporation. *Intel oneAPI Programming Guide*. https://community.intel.com/legacyfs/online/drupal_files/oneAPIProgrammingGuide_9.pdf. 2023.

[149] *Level Zero Specification documentation*. https://spec.oneapi.io/level-zero/latest/core/INTRO.html.

[150] *Intel® DPC++ Compatibility Tool*. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compatibility-tool.html (visitado 26-12-2023).

[151] S. Christgau y T. Steinke. «Porting a Legacy CUDA Stencil Code to oneAPI». En: *2020 IEEE IPDPSW*. 2020, págs. 359-367. DOI: 10.1109/IPDPSW50202.2020.00070.

[152] Yuhsiang M. Tsai, Terry Cojean y Hartwig Anzt. *Porting a sparse linear algebra math library to Intel GPUs*. 2021. arXiv: 2103.10116 [cs.DC].

[153] Pablo Antonio Martínez et al. «Applying Intel's oneAPI to a machine learning case study». En: *Concurrency and Computation: Practice and Experience* 34.13 (2022), e6917. DOI: https://doi.org/10.1002/cpe.6917. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.6917. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6917.

[154] Youssef Faqir-Rhazoui y Carlos García. «Exploring the Performance and Portability of the K-Means Algorithm on SYCL across CPU and GPU Architectures». En: *J. Supercomput.* 79.16 (mayo de 2023), págs. 18480-18506. ISSN: 0920-8542. DOI: 10.1007/s11227-023-05373-2. URL: https://doi.org/10.1007/s11227-023-05373-2.

[155] Zheming Jin y Jeffrey Vetter. «Evaluating CUDA Portability with HIPCL and DPCT». En: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2021, págs. 371-376. DOI: 10.1109/IPDPSW52791.2021.00065.

[156] Germán Castaño et al. «Evaluation of Intel's DPC++ Compatibility Tool in heterogeneous computing». En: *Journal of Parallel and Distributed Computing* 165 (2022), págs. 120-129. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2022.03.017. URL: https://www.sciencedirect.com/science/article/pii/S0743731522000727.

[157] Wang Yong et al. «Developing Medical Ultrasound Imaging Application across GPU, FPGA, and CPU Using OneAPI». En: *International Workshop on OpenCL*. IWOCL'21. Munich, Germany: Association for Computing Machinery, 2021. ISBN: 9781450390330. DOI: 10.1145/3456669.3456680. URL: https://doi.org/10.1145/3456669.3456680.

[158] Eugenio Marinelli y Raja Appuswamy. «XJoin: Portable, Parallel Hash Join across Diverse XPU Architectures with OneAPI». En: *Proceedings of the 17th International Workshop on Data Management on New Hardware*. DAMON '21. Virtual Event, China: Association for Computing Machinery, 2021. ISBN: 9781450385565. DOI: 10.1145/3465998.3466012. URL: https://doi.org/10.1145/3465998.3466012.

[159] Zheming Jin y Jeffrey S. Vetter. «Understanding Performance Portability of Bioinformatics Applications in SYCL on an NVIDIA GPU». En: *2022 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. 2022, págs. 2190-2195. DOI: 10.1109/BIBM55620.2022.9995222.

[160] Muhammad Haseeb et al. «Evaluating Performance and Portability of a core bioinformatics kernel on multiple vendor GPUs». En: *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 2021, págs. 68-78. DOI: 10.1109/P3HPC54578.2021.00010.

[161] Leonardo Solis-Vasquez, Edward Mascarenhas y Andreas Koch. «Experiences Migrating CUDA to SYCL: A Molecular Docking Case Study». En: *Proceedings of the 2023 International Workshop on OpenCL*. IWOCL '23. Cambridge, United Kingdom: Association for Computing Machinery, 2023. DOI: 10.1145/3585341.3585372. URL: https://doi.org/10.1145/3585341.3585372.

[162] Eugenio Marinelli y Raja Appuswamy. «OneJoin: Cross-architecture, scalable edit similarity join for DNA data storage using oneAPI». En: *ADMS 2021, 12th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, in conjunction with VLDB 2021, 16 August 2021, Copenhagen, Denmark*. Ed. por ACM. Copenhagen, 2021.

[163] Om Gangadhar Jadhav et al. «Migration of a Cuda Based Seismic Modeling Application to Sycl Codebase for Heterogeneous Architectures». En: *Available at SSRN 4515781* (2023).

[164] Zheming Jin. *Experience of Migrating Parallel Graph Coloring from CUDA to SYCL*. Inf. téc. Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2022.

[165] Enzo Rucci et al. «SWIFOLD: Smith-Waterman implementation on FPGA with OpenCL for long DNA sequences». En: *BMC systems biology* 12.Suppl 5 (nov. de 2018), pág. 96. ISSN: 1752-0509. DOI: 10.1186/s12918-018-0614-6. URL: https://europepmc.org/articles/PMC6245597.

[166] NVIDIA. *Nsight Compute*. https://developer.nvidia.com/nsight-compute. 2022.

[167] *Kernel Profiling Guide*. Sep. de 2023. URL: https://docs.nvidia.com/nsight-compute/ProfilingGuide/.

[168] MJ Rutter. *Intel's Variable Clock Speeds and Benchmarking*. https://www.mjr19.org.uk/IT/clocks.html. 2023.

[169] Intel Corporation. *Alder Lake S*. https://www.intel.la/content/www/xl/es/products/platforms/details/alder-lake-s.html. 2023.

[170] Brian Homerding y John Tramm. «Evaluating the Performance of the HipSYCL Toolchain for HPC Kernels on NVIDIA V100 GPUs». En: *Proceedings of the International Workshop on OpenCL*. IWOCL '20. Munich, Germany: Association for Computing Machinery, 2020. ISBN: 9781450375313. DOI: 10.1145/3388333.3388660. URL: https://doi.org/10.1145/3388333.3388660.

[171] Istvan Z Reguly. «Evaluating the performance portability of SYCL across CPUs and GPUs on bandwidth-bound applications». En: *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 2023, págs. 1038-1047.

[172] Esteban Miguel Rangel et al. «A Performance-Portable SYCL Implementation of CRK-HACC for Exascale». En: *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 2023, págs. 1114-1125.

[173] Julian Franquinet. «Performance portability analysis of SYCL with a classical CG on CPU, GPU, and FPGA». Tesis de mtría. 2023.

[174] Phuong Nguyen, Pratik Nayak y Hartwig Anzt. «Porting Batched Iterative Solvers onto Intel GPUs with SYCL». En: SC-W '23. ¡conf-loc¿, ¡city¿Denver¡/city¿, ¡state¿CO¡/state¿, ¡country¿USA¡/country¿, ¡/conf-loc¿: Association for Computing Machinery, 2023, págs. 1048-1058. DOI: 10.1145/3624062.3624181. URL: https://doi.org/10.1145/3624062.3624181.

[175] Rocío Carratalá-Sáez et al. «Open SYCL on heterogeneous GPU systems: A case of study». En: *arXiv preprint arXiv:2310.06947* (2023).

[176] Zheming Jin y Jeffrey S Vetter. «Understanding Performance Portability of SYCL Kernels: A Case Study with the All-Pairs Distance Calculation in Bioinformatics on GPUs». En: *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2023, págs. 366-372.

[177] Zheming Jin y Jeffrey S Vetter. «Understanding SYCL Portability for Pseudorandom Number Generation: a Case Study with Gene-Expression Connectivity Mapping». En: *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2023, págs. 295-298.

[178] Christoph Weckert et al. «Altis-SYCL: Migrating Altis Benchmarking Suite from CUDA to SYCL for GPUs and FPGAs». En: *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 2023, págs. 547-555.

[179] Leonard Apanasevich et al. «A Comparison of the Performance of the Molecular Dynamics Simulation Package GROMACS Implemented in the SYCL and CUDA Programming Models». En: (2023).

[180] Beau Johnston, Jeffrey S. Vetter y Josh Milthorpe. «Evaluating the Performance and Portability of Contemporary SYCL Implementations». En: *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 2020, págs. 45-56. DOI: 10.1109/P3HPC51967.2020.00010.

[181] Marcel Breyer, Gregor Daiß y Dirk Pflüger. «Performance-Portable Distributed k-Nearest Neighbors Using Locality-Sensitive Hashing and SYCL». En: *International Workshop on OpenCL*. IWOCL'21. Munich, Germany: Association for Computing Machinery, 2021. ISBN: 9781450390330. DOI: 10.1145/3456669.3456692. URL: https://doi.org/10.1145/3456669.3456692.

[182] Wageesha R. Shilpage y Steven A. Wright. «An Investigation into the Performance and Portability of SYCL Compiler Implementations». En: *High Performance Computing*. Ed. por Amanda Bienz et al. Cham: Springer Nature Switzerland, 2023, págs. 605-619. ISBN: 978-3-031-40843-4.