Análisis de la portabilidad de la implementación de métodos numéricos de hidrodinámica de partículas suaves en diferentes plataformas y frameworks CPU/GPU

Fabian Arturo Sánchez Calderón, Wilmer Steven Farfán Castillo

Trabajo de Grado para Optar al Título de Ingeniero de Sistemas

Director

MSc. Sergio Augusto Gélvez Cortés

Magister en Ingeniería de Sistemas

Asesor

PhD. Carlos Jaime Barrios Hernández

Doctor en Informática y Ciencias Computacionales

Universidad Industrial de Santander

Facultad De Ingenierías Fisicomecánicas

Escuela De Ingeniería De Sistemas E Informática

Bucaramanga Colombia

2025

Dedicatoria

Este proyecto va dedicado a nuestros padres, que sin su apoyo incondicional esta preparación universitaria no hubiese sido posible. A compañeros y profesores que nos dieron su apoyo durante esta bonita etapa de la vida.

Y finalmente a nosotros mismos, por el esfuerzo titánico realizado durante todos estos años de vida universitaria. Por la superación personal, la preparación académica y técnica, la mejora de habilidades, el crecimiento como profesionales, el sorteamiento de muchas dificultades y la motivación más grande; culminar nuestros estudios universitarios.

Agradecimientos

Agradecimientos en primer lugar a la Universidad Industrial de Santander, por ser el claustro que brindó la oportunidad de adquirir conocimientos, vivir nuevas experiencias, conocer personas, aprender muchas cosas paralelas a la academia y ser la impulsora hacia un buen futuro profesional.

A nuestro director del proyecto MSc. Sergio Augusto Gélvez Cortés por su valiosa guía durante el desarrollo del presente proyecto. Al profesor PhD. Carlos Jaime Barrios Hernández por su orientación y dirección durante las primeras fases del proyecto y al grupo SC3 por sus aportes en materia de infraestructura para este trabajo.

Tabla de Contenido

Lista de Tablas

Pág.

Lista de Figuras

Pág.

Glosario

Kernels: Funciones que se ejecutan en la GPU en paralelo sobre múltiples hilos para procesar un gran conjunto de datos.

Host–Device: Modelo de cómputo que distingue entre la CPU (host) y la GPU (device) como espacios de ejecución y memoria separados.

Streaming asíncrono: Ejecución de operaciones en GPU sin bloquear la CPU, permitiendo el solapamiento de cómputo y transferencia de datos.

Solapamiento: Estrategia que combina cómputo y comunicación simultáneamente para reducir los tiempos de espera.

Multi-GPU: Uso coordinado de varias tarjetas gráficas en paralelo para aumentar la capacidad de cómputo.

Grid–Block–Thread: Jerarquía de organización en CUDA/HIP dónde los hilos se agrupan en bloques y los bloques en una grilla.

Toolchain: Conjunto de compiladores, enlazadores y utilidades necesarias para construir y ejecutar un programa.

CUDA Toolkit: Conjunto oficial de librerías, compiladores y herramientas de Nvidia para desarrollar aplicaciones en GPU.

Hipificación: Proceso de traducir código CUDA a HIP para permitir su ejecución en GPUs AMD u otras plataformas.

***cudaMemcpy*:** Función de CUDA que transfiere datos entre memoria del host y memoria del device.

hipMemcpy: Equivalente en HIP a *cudaMemcpy*, utilizada para copiar datos entre CPU y GPU.

Device Linker: Enlazador especializado que combina múltiples objetos con código de GPU en un único ejecutable.

hipMemcpyToSymbol: Función HIP que copia datos desde el host hacia una variable constante en memoria de GPU.

Wrappers: Interfaces que encapsulan funciones o librerías para facilitar su uso en otros lenguajes o entornos.

nodiscard: Especificador de C++ que indica que el valor de retorno de una función no debe ser ignorado.

Módulo dedicado: Componente de software diseñado exclusivamente para una tarea específica dentro del sistema.

FMA (Fused Multiply-Add): Instrucción que combina en una sola operación la multiplicación y la suma ($a \times b + c$), con un único redondeo final; mejora el rendimiento pero puede generar ligeras discrepancias numéricas.

Full GPU: Hace referencia a los kernels CUDA que implementan los cálculos principales del método SPH: construcción de búsqueda de partículas vecinas, interacción de partículas (fuerzas de presión y viscosidad) y actualización del estado (posición, velocidad y densidad de las partículas).

TLC (Cobertura de Traducción Automática): Indicador que mide el alcance de las porciones de código traducidas automáticamente por herramientas como HIPIFY, sin intervención manual.

MIB (Carga de Integración Manual): Estimación del esfuerzo requerido en tareas de adaptación manual del código, incluyendo ajustes de cabeceras, namespaces y funciones no soportadas.

TSI (Índice de Déficit de Traducción): Métrica que cuantifica la proporción de elementos no traducidos o incompatibles tras el proceso automático, reflejando las brechas de portabilidad.

BCP (Progreso de Compilación): Valor que representa el avance alcanzado en la construcción del binario final, considerando tanto fases exitosas como errores de enlace o dependencia.

Resumen

Título: Análisis de la portabilidad de la implementación de métodos numéricos de hidrodinámica de partículas suaves en diferentes plataformas y frameworks CPU/GPU.[1*]

Autor: Fabian Arturo Sánchez Calderón, Wilmer Steven Farfán Castillo.[2*]

Palabras Clave: SPH, portabilidad, GPU, CUDA, ROCm, HIP, HPC, simulación numérica.

Descripción: La hidrodinámica de partículas suaves (SPH) es un método consolidado para el modelado de fluidos u otros materiales deformables, que dada a su naturaleza exigen arquitecturas de alto rendimiento (HPC) para su ejecución, en especial aquellas aceleradas por GPU (Yang et al., 2024). Este proyecto aborda el análisis de la portabilidad de implementaciones numéricas de SPH en diferentes frameworks y plataformas CPU/GPU, tomando como punto de partida el entorno CUDA de Nvidia, que ha demostrado mejoras muy notables en la eficiencia en la ejecución de sus algoritmos (Monaghan, 2005), así como alternativas abiertas como ROCm/HIP de AMD y otros modelos como OpenMP. Se evaluó el proceso de traducción entre plataformas mediante HIPIFY, una herramienta que convierte código CUDA a HIP. Sin embargo, un programa optimizado para una GPU de un proveedor en específico puede que no ofrezca una ejecución eficiente para otros fabricantes (Krog & Elster, 2010). En este enfoque se evidencian limitaciones relacionadas la falta de soporte de ciertos intrínsecos, el uso de constructos particulares de CUDA y las discrepancias existentes de las bibliotecas entre ambas tecnologías, que obliga una intervención manual para garantizar su ejecución. El análisis permitió identificar los principales desafíos técnicos en términos de compatibilidad, rendimiento y escalabilidad, donde si bien es posible alcanzar niveles funcionales de adaptación, esto conlleva un costo adicional en ajustes de código y optimización del mismo. Finalmente, se presentan recomendaciones orientadas a mejorar la adaptabilidad y la eficiencia de futuras implementaciones SPH.

[1*] Trabajo de Grado

[2**] Facultad de Ingenierías Fisicomecánicas. Escuela De Ingeniería De Sistemas e Informática. Director: MSc. Sergio Augusto Gélvez Cortés. Magister en Ingeniería de Sistemas. Asesor: PhD. Carlos Jaime Barrios Hernández. Doctor en Informática y Ciencias Computacionales.

Abstract

Title: Analysis of the Portability of the Implementation of Smoothed Particle Hydrodynamics Numerical Methods across Different CPU/GPU Platforms and Frameworks.[3*]

Author: Fabian Arturo Sánchez Calderón, Wilmer Steven Farfán Castillo[4]

Key Words: SPH, portability, GPU, CUDA, ROCm, HIP, HPC, numerical simulation.

Description: Smooth Particle Hydrodynamics (SPH) is a consolidated method for modeling fluids or other deformable materials, which given its nature, requires high-performance computing (HPC) architectures for its execution, especially those accelerated by GPU (Yang et al., 2024). This project addresses the analysis of the portability of numerical implementations of SPH across different frameworks and CPU/GPU platforms, taking as a starting point Nvidia's CUDA environment, which has demonstrated very notable improvements in the efficiency of algorithm execution (Monaghan, 2005), as well as open alternatives such as AMD's ROCm/HIP and other models such as OpenMP. The translation process between platforms was evaluated through HIPIFY, a tool that converts CUDA code to HIP. However, a program optimized for a GPU from a specific vendor may not provide efficient execution for other manufacturers (Krog & Elster, 2010). In this approach, limitations become evident regarding the lack of support for certain intrinsics, the use of CUDA-specific constructs and the discrepancies present in the libraries between both technologies, which requires manual intervention to ensure execution. The analysis made it possible to identify the main technical challenges in terms of compatibility, performance and scalability, where although it is possible to achieve functional levels of adaptation, this entails an additional cost in code adjustments and optimization. Finally, recommendations are presented aimed at improving the adaptability and efficiency of future SPH implementations.

[3*] Degree Work

[4**] Faculty of Mechanical and Physical Engineering. School of Systems and Computer Science. Supervisor: M.Sc. Sergio Augusto Gélvez Cortés. Magister in Systems Engineering. Advisor: Ph.D. Carlos Jaime Barrios Hernández. Doctor in Computer Science and Computational Sciences..

Introducción

La hidrodinámica de partículas suaves (*Smoothed Particle Hydrodynamics, SPH*) se ha consolidado como una herramienta alternativa, para la investigación de problemas científicos y la resolución de problemas prácticos de ingeniería, motivado por el rápido desarrollo del hardware informático y la rápida mejora del rendimiento computacional (Long S, et al., 2023). En este contexto, el presente proyecto se centra en el análisis de la portabilidad de la implementación de métodos numéricos de SPH en diversas plataformas y frameworks CPU/GPU.

Los métodos SPH ofrecen una herramienta versátil para modelar el comportamiento de fluidos y materiales deformables. No obstante, el alto costo computacional del SPH hace que las arquitecturas de computación de alto rendimiento (HPC), en especial los aceleradores de gráficos (GPU), sean un componente fundamental para la simulación de fenómenos físicos complejos (Yang Q, et al., 2024). Actualmente, el uso de plataformas aceleradas por GPU, como CUDA de NVIDIA, ha permitido mejoras significativas en el rendimiento y la eficiencia computacional de estas implementaciones SPH (Monaghan J, 2005).

Sin embargo, la portabilidad entre diferentes plataformas y frameworks CPU/GPU se ha convertido en un desafío en donde la diversidad de arquitecturas de hardware y la coexistencia de múltiples toolchains (CUDA, ROCm, OpenMP, SYCL) dificulta la transferencia eficiente de implementaciones optimizadas entre distintas plataformas, limitando la flexibilidad y escalabilidad de las aplicaciones. Un programa que ha sido optimizado para una GPU de un

proveedor puede que no tenga una ejecución eficiente en la próxima generación de procesadores o en un dispositivo de un fabricante diferente (Krog Ø, Elster A, 2010).

En el ecosistema ROCm de AMD, HIPIFY *(especialmente hipify-clang*) traduce fuentes CUDA a lenguaje HIP y este a su vez reporta constructos no cubiertos (*intrinsics, cooperative groups, texturas/arrays, usos específicos de bibliotecas CUDA*), por lo que la traducción funciona como un punto de partida y requiere de una intervención manual (AMD ROCm HIPIFY, s. f.).

Este trabajo aborda la problemática mediante un análisis comparativo de la portabilidad de las implementaciones SPH entre diferentes frameworks CPU/GPU, identificando ventajas, desventajas y métricas de rendimiento. Este enfoque permitirá formular recomendaciones para futuros desarrollos en este campo, busca dimensionar su costo real y sus implicaciones en mantenimiento, validación y evolución a la luz de métricas aceptadas en HPC, con el objetivo de mejorar la flexibilidad y eficiencia de las aplicaciones de simulación de fluidos y materiales deformables.

Planteamiento y Justificación

La portabilidad de las implementaciones de métodos numéricos de hidrodinámica de partículas suaves (SPH) en diversas plataformas y frameworks CPU/GPU representa un obstáculo significativo en la comunidad científica y la industria. La falta de portabilidad limita la flexibilidad y escalabilidad de las aplicaciones SPH, restringiendo su potencial para abordar problemas complejos en diversos campos como la ingeniería, la física computacional y la simulación de fluidos.

Este proyecto aborda el problema de la portabilidad de las implementaciones SPH mediante un análisis de las ventajas y desventajas de diferentes plataformas y frameworks CPU/GPU. El objetivo es identificar las mejores prácticas y formular recomendaciones para futuros desarrollos que mejoren la portabilidad y eficiencia de las aplicaciones SPH.

Se espera que este proyecto de grado contribuya a mejorar la comprensión de los desafíos y oportunidades relacionados con la portabilidad de las implementaciones SPH, a través de la identificación de las mejores prácticas para la implementación de métodos SPH en diferentes plataformas y frameworks CPU/GPU, así como el desarrollo de estrategias para mejorar su portabilidad.

1. Objetivos

1.1. Objetivo General

Analizar la portabilidad de la implementación de métodos numéricos de hidrodinámica de partículas suaves (SPH) en diversas plataformas y frameworks CPU/GPU con el objetivo de identificar sus puntos fuertes y débiles, así como los retos y dificultades que se presentan en términos de portabilidad y rendimiento.

1.2. Objetivos Específicos

- Realizar un análisis de la implementación existente en CUDA, identificando los aspectos clave de su diseño y rendimiento en esta plataforma.
- Desarrollar una versión adaptada de la implementación para la plataforma ROCm de AMD, aprovechando las herramientas y recursos proporcionados por este entorno.

- Realizar pruebas de rendimiento para comparar el desempeño en CUDA y ROCm, utilizando métricas relevantes como el tiempo de ejecución, la utilización de recursos y la eficiencia computacional.
- Identificar y analizar los desafíos y limitaciones encontrados durante el proceso de adaptación a ROCm.
- Formular recomendaciones para optimizar la adaptabilidad y rendimiento basándose en los resultados obtenidos y los desafíos identificados.

2. Marco de Referencia

A continuación, se abordan los conceptos fundamentales necesarios para comprender el problema de la portabilidad de las implementaciones de métodos numéricos de hidrodinámica de partículas suaves (SPH) en diversas plataformas y frameworks CPU/GPU. A continuación, se presentan los conceptos necesarios para cumplir con los objetivos propuestos.

2.1. Métodos Numéricos de Hidrodinámica de Partículas Suaves (SPH):

La hidrodinámica de partículas suaves o suavizadas (*Smoothed Particle Hydrodynamics*) es un método de partículas lagrangiano sin malla que ofrece ventajas sobre los métodos numéricos convencionales basados en cuadrículas en el tratamiento de interfaces. El método se caracteriza por su sencillez y generalidad, posibilitando el tratamiento de flujos de superficie libre con muy pocas restricciones. (Grassa, 2004).

El método ha demostrado un gran potencial para la simulación de problemas de flujo multifásico, mediante el uso de una serie de partículas que transportan propiedades físicas para

representar fluidos continuos (densidad, velocidad, presión). Estas partículas interactúan entre sí a través de una función de interpolación suave, lo que permite simular el comportamiento de fluidos complejos y materiales deformables. (Monaghan J, 2005).

A partir de estas partículas, se obtienen soluciones numéricas aproximadas mediante aproximaciones de núcleo (*kernel approximation*) y de partículas, que constituyen los pasos más importantes en SPH. La aproximación de núcleo emplea una representación integral ponderada para la aproximación de la función de campo, donde la función de ponderación utilizada en la representación integral se denomina función de núcleo de suavizado (*smoothing kernel function*) (Wang et al., 2016),

$$f(r) = \int_{\Omega} f(r')\delta(r - r')dr'$$

donde dr' denota un elemento de volumen, $f(r)$ es una función en la posición r, h es la longitud del suavizado y r representa el dominio integral. La función delta se reemplaza por una función de suavizado y se simplifica por:

$$f(r) \approx \int_{\Omega} f(r')W(r - r',\ h)dr'$$

La aproximación de partículas se realiza sustituyendo la representación integral de la función de campo por la suma de todos los valores correspondientes en las partículas vecinas. Como los fluidos son representados por un conjunto de partículas en SPH, cada partícula *j* tiene masa m_j , densidad p_j y posición r_j la aproximación es:

$$f(r_i) \approx \sum_{j=1}^{N} f(r_j)W(r_i - r_j,\ h)\ \frac{m_j}{p_j},$$

y el gradiente de la función se calcula como:

$$\nabla f(r_i) \approx \sum_{j=1}^{N} \frac{m_j}{p_j} f(r_j) \nabla W(r_i - r_j, h),$$

donde N es el número de partículas dentro de la región de soporte del núcleo de la partícula i. (Wang et al., 2016).

2.2. SPhysics:

Es un software de código abierto para simulaciones fluidodinámicas basado en métodos SPH. Proporciona una plataforma flexible para modelar una amplia gama de fenómenos fluidodinámicos, incluidos flujos complejos de fluidos y estructuras deformables. Nació como un proyecto colaborativo entre la Universidad Johns Hopkins, la Universidad de Vigo y la Universidad de Manchester, escrito inicialmente en Fortran 90, con distintos kernels y condiciones de frontera. Este software fué la base a partir de la cual se creó **DualSPHysics** para un uso más eficiente en GPUs y HPC (DualSPHysics, 2024). SPHysics se ha utilizado en una variedad de campos de investigación, como oceanografía, ingeniería civil y biología, debido a su capacidad para manejar problemas complejos de flujo y transporte. (Domínguez, et al., 2015).

2.2.1. DualSPhysics :

DualSPhysics es un solucionador de las ecuaciones de Navier–Stokes mediante el método de hidrodinámica de partículas suavizadas (SPH). DualSPHysics está implementado en C++ y CUDA, permitiendo realizar simulaciones con millones de partículas en CPU o GPU, respectivamente, ofreciendo como ventajas un uso más optimizado de la memoria. CUDA gestiona la ejecución paralela de hilos en las GPUs.

Bajo el paradigma de la programación orientada a objetos, DualSPhysics proporciona un código que se permite modificar con un control sofisticado de errores. Además, implementan

optimizaciones más avanzadas como por ejemplo, las partículas se reordenan para un acceso más rápido a la memoria y se aplica el mejor método para crear la lista de vecinos (DualSPHysics, s.f.).

2.3. Computación de Alto Rendimiento (HPC):

La computación de alto rendimiento (HPC) es una tecnología que utiliza clústeres de potentes procesadores que trabajan en paralelo para procesar conjuntos de datos multidimensionales masivos (big data) y para resolver problemas complejos a velocidades extremadamente altas. La computación paralela ejecuta múltiples tareas de manera simultánea en varios servidores o procesadores informáticos. Un clúster de HPC consta de varios servidores informáticos de alta velocidad conectados en red, con un planificador centralizado que administra la carga de trabajo de computación paralela. (IBM, s.f.).

HPC resuelve algunos de los problemas informáticos actuales más complejos en tiempo real, se busca maximizar el rendimiento bajo restricciones de memoria, datos de entrada/salida, energía, que incluye el uso de supercomputadoras, clústeres de computadoras y arquitecturas aceleradas por GPU. (Hager & Wellein, 2011).

2.4. Plataformas Aceleradas por GPU:

Las unidades de procesamiento gráfico (GPU) constituyen un ecosistema de hardware y software, diseñado para ser utilizado en el paralelismo de tareas de propósito general. Las GPU maximizan el rendimiento por medio del uso de hilos ligeros bajo el paradigma SIMT, permitiendo realizar una misma operación en grandes lotes de datos. Las GPUs se utilizan cada vez más en aplicaciones de cómputo de alto rendimiento debido a su capacidad para realizar

operaciones en paralelo a gran escala. Por medio de plataformas como CUDA de Nvidia y ROCm de AMD, se ofrecen entornos de programación especializados en la optimización de las GPU para sus diversas aplicaciones científicas y de ingeniería.

2.5. OpenMP:

OpenMP es una interfaz de programación de aplicaciones (API) para la programación multiproceso de memoria compartida en múltiples plataformas. La API de OpenMP admite la programación paralela en memoria compartida y multiplataforma en C/C++ y Fortran. Esta API define un modelo portátil y escalable con una interfaz simple y flexible para el desarrollo de aplicaciones paralelas en plataformas que van desde equipos de escritorio hasta supercomputadoras. La API de OpenMP cubre únicamente la paralelización dirigida por el usuario, en la cual el programador especifica explícitamente las acciones que deben realizar el compilador y el sistema de ejecución para ejecutar el programa en paralelo (Lewis, 2025).

La API de OpenMP utiliza el modelo de ejecución paralelo *fork-join*, diseñado especialmente para aplicaciones con grandes arreglos. Un programa en OpenMP puede ejecutarse tanto en modo paralelo (*c*on múltiples hilos y soporte completo de la librería) como en modo secuencial (ignorando directivas). Sin embargo, es posible que un programa produzca resultados diferentes entre su ejecución secuencial y paralela, debido a las variaciones en la asociación de operaciones numéricas, como lo serían por ejemplo en reducciones de suma en punto flotante (Lewis, 2025).

La ejecución comienza con un único hilo inicial. Cuando este encuentra una construcción paralela, crea un equipo de hilos (actuando como maestro) que ejecuta conjuntamente la región paralela. Al final de cada región paralela hay una barrera implícita: solo el hilo maestro continúa

más allá. Las regiones paralelas pueden anidarse. En los constructos de reparto de trabajo, las tareas se dividen entre los hilos del equipo en lugar de ser ejecutadas por todos.

Existen también constructos de sincronización y rutinas de biblioteca para coordinar hilos y datos, así como variables de entorno que permiten controlar el tiempo de ejecución. OpenMP no garantiza sincronización en operaciones de entrada/salida sobre un mismo archivo cuando se ejecuta en paralelo; en este caso, corresponde al programador asegurar la sincronización (Lewis, 2025).

Las implementaciones conformes con OpenMP no están obligadas a verificar dependencias de datos, conflictos de datos, condiciones de carrera o interbloqueos. Tampoco están obligadas a comprobar secuencias de código que provoquen que un programa sea clasificado como no conforme. Es responsabilidad de los desarrolladores de las aplicaciones el utilizar correctamente la API de OpenMP para producir un programa conforme. La API de OpenMP no contempla la paralelización automática generada por el compilador. (Lewis, 2025)

2.6. CUDA (*Compute Unified Device Architecture*):

Es una plataforma de computación paralela y modelo de programación desarrollada por Nvidia para ejecutar diversas operaciones de propósito general sobre sus GPUs. Para organizar la ejecución, CUDA maneja una jerarquía compuesta por cuadrículas (*grid*), que integran bloques (*blocks*) de hilos (*threads*), formados por máximo 1024 hilos distintos. facilita la asignación de miles de hilos directamente a los recursos físicos disponibles en la GPU, para así realizar las operaciones de cálculo requeridas.

Proporciona un entorno de desarrollo, con un compilador y una API que permite a desarrolladores e investigadores escribir código para ejecutar diversas tareas. Estas se ejecutan

haciendo uso de los núcleos CUDA, que son unidades de procesamiento presentes en las GPUs de Nvidia más recientes, diseñadas para realizar operaciones básicas y trabajar en conjunto por medio del paralelismo que ofrecen sus múltiples núcleos (Nvidia, 2025). De esta manera, se convierte a las GPUs en plataformas de cómputo acelerado. CUDA ha sido ampliamente adoptado en aplicaciones de computación de alto rendimiento y es compatible con una variedad de lenguajes de programación, incluidos C, C++, y Python. (Sanders & Kandrot, 2010).

El código de la GPU se implementa como una colección de funciones en un lenguaje que es esencialmente C++, pero con algunas anotaciones adicionales que permiten distinguirlas del código anfitrión y para diferenciar los distintos tipos de memoria de datos existentes en la GPU. Estas funciones pueden tener parámetros, se llaman usando una sintaxis muy similar a la de una función normal en C, aunque ligeramente extendida para poder especificar la malla de hilos de GPU que deben ejecutar la función invocada. Durante su ciclo de vida, el proceso anfitrión puede despachar múltiples tareas paralelas a la GPU (Nvidia, 2025).

También es posible utilizar otros lenguajes como Fortran, Julia y Java, por medio de "wrappers" que actúan como una capa de software intermedia invocando rutinas de bajo nivel escritas en C/C++.

2.7. NVCC (*Nvidia Cuda Compiler*):

NVCC es un compilador de Nvidia que está diseñado para usarse con CUDA. El código CUDA se ejecuta tanto CPU como en GPU, NVCC separa ambas partes y envía el código anfitrión, lo que se ejecutará en la CPU, a un compilador de C o C++ como GCC, Intel C++ o MVCC. El código del dispositivo, denominado *kernels,* lo ejecutará directamente sobre la GPU.

La trayectoria de compilación implica varios pasos de separación, compilación, preprocesamiento y unión para cada archivo fuente CUDA. El propósito de *nvcc*, el controlador del compilador CUDA, es ocultar los detalles complejos de la compilación CUDA a los desarrolladores. Acepta una variedad de opciones convencionales de compilador, como la definición de macros, rutas de inclusión/bibliotecas y directivas para controlar el proceso de compilación. Todos los pasos de compilación no relacionados con CUDA se reenvían a un compilador C++ anfitrión soportado por NVCC, y este traduce sus opciones en comandos adecuados para dicho compilador (Nvidia, 2025).

Los archivos fuente de las aplicaciones CUDA consisten en una mezcla de código anfitrión en C++ convencional y funciones de dispositivo GPU. La trayectoria de compilación CUDA separa las funciones de dispositivo del código anfitrión, compila las funciones de dispositivo utilizando compiladores y ensambladores de Nvidia, compila el código anfitrión con un compilador C++ disponible y posteriormente incrusta las funciones compiladas de GPU como imágenes *fatbinary* dentro del archivo objeto anfitrión. En la etapa de enlace, se añaden bibliotecas específicas de tiempo de ejecución CUDA para soportar llamadas remotas a procedimientos SPMD y proporcionar operaciones de manipulación de la GPU, como la asignación de buffers de memoria y la transferencia de datos entre CPU y GPU.

La trayectoria de compilación tiene que pasar por una serie de pasos de separación, compilación, preprocesamiento y unión para cada archivo CUDA. El propósito de NVCC es facilitar el proceso de compilación sin involucrar al desarrollador con los detalles complejos de la compilación CUDA. Acepta una variedad de opciones de compilador, como la definición de macros, rutas de inclusión/bibliotecas y directivas para controlar este proceso. Todos los pasos de

compilación no relacionados con CUDA se remiten a un compilador C++ anfitrión soportado por NVCC , para la cual este lo traduce en comandos soportados por dicho compilador.

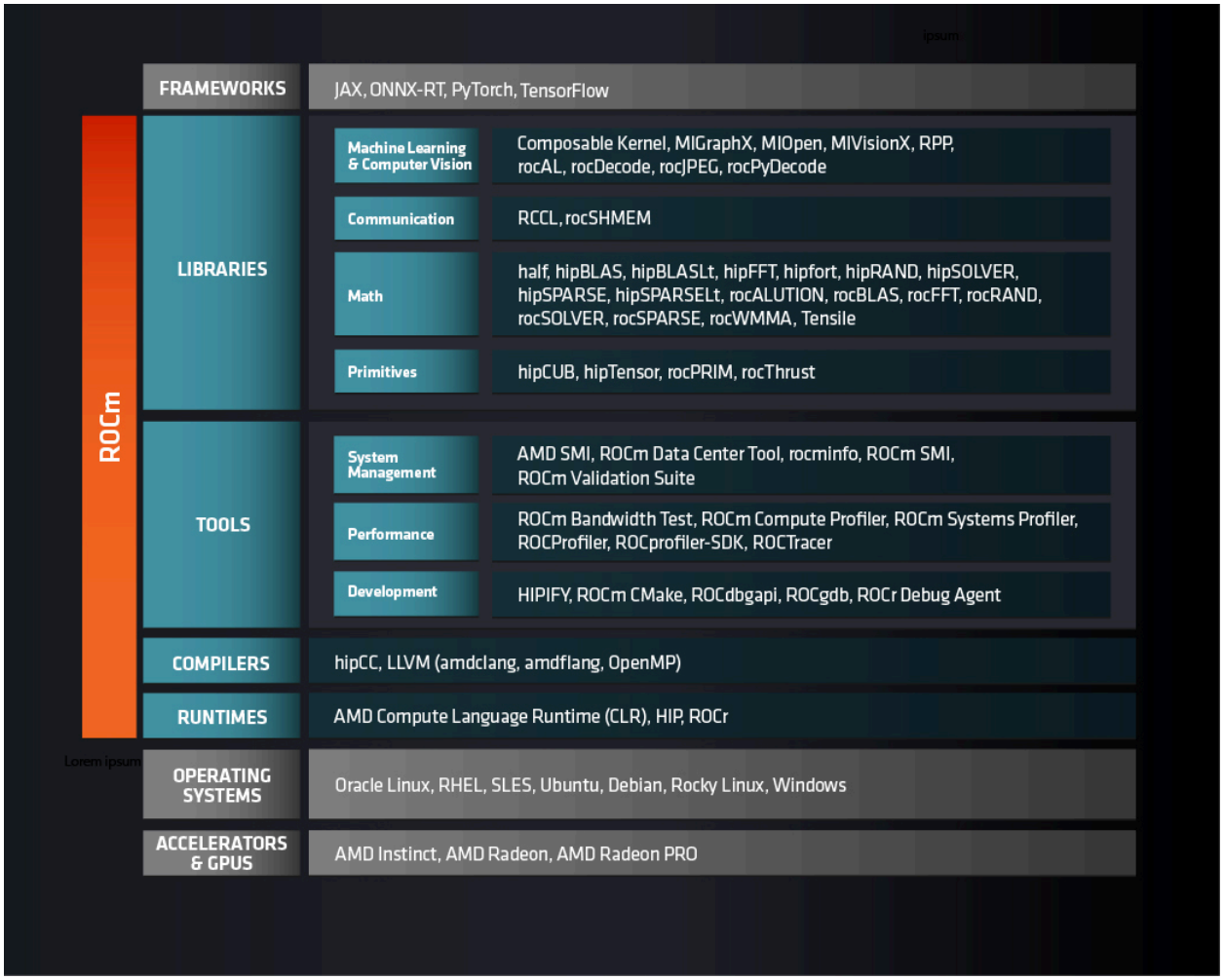
2.8. ROCm:

ROCm es una plataforma de software de código abierto desarrollada por AMD, que incluye compiladores, bibliotecas para funciones de alto nivel, depuradores, perfiladores y entornos de ejecución, facilitando la programación de GPU desde el kernel de bajo nivel hasta las aplicaciones de usuario final. Está optimizada para extraer el máximo rendimiento de cargas de trabajo de HPC e IA a partir de los aceleradores AMD Instinct y las GPU AMD Radeon, manteniendo al mismo tiempo la compatibilidad con los marcos de software de la industria (ROCm, s.f.).

ROCm admite múltiples lenguajes e interfaces de programación, como HIP (Heterogeneous-Compute Interface for Portability), OpenCL y OpenMP. No obstante a diferencia de CUDA, no se encuentra fuertemente ligado al ecosistema propietario, sino que ofrece un enfoque multiplataforma con soporte para diversos compiladores. ROCm proporciona herramientas y bibliotecas para la programación de GPU utilizando lenguajes como C++ y Python, además ofrece soporte para la interoperabilidad de GPU entre diferentes frameworks y lenguajes de programación. (ROCm, s.f.).

Figura 1.

Estructura de ROCm. (ROCm, s.f.).



En la Figura 1, se puede apreciar el ecosistema que compone a ROCm y su amplia compatibilidad con diversos frameworks muy utilizados en la industria, además de permitir su ejecución en varias distribuciones de Linux y Windows. Para ser ejecutado correctamente ROCm requiere de ciertos modelos específicos de GPU debido a que la pila está pensada y validada para un subconjunto específico de firmware, que cuentan con las capacidades de cómputo requeridas para su uso en HPC.

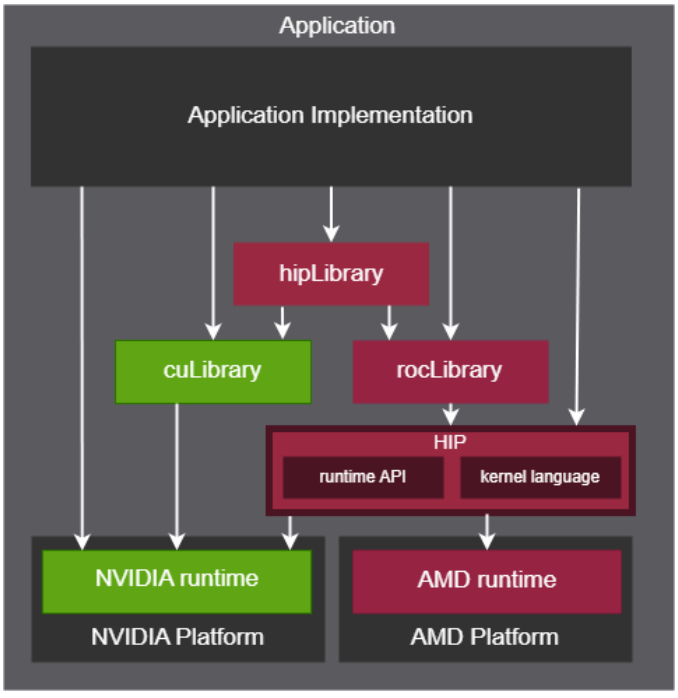
2.9. HIP (*Heterogeneous-Compute Interface for Portability*):

HIP es una API en tiempo de ejecución de C++ y un lenguaje de kernel, que permite a los desarrolladores crear aplicaciones portables para GPUs de AMD y Nvidia a partir de un único código fuente. Proporciona una síntesis muy similar a CUDA sin ser una emulación y busca posicionarse como una capa intermedia que busca facilitar la portabilidad, sin sacrificar rendimiento. De esta manera se busca reducir el esfuerzo de migración y facilitar la ejecución nativa en diversas GPUs. (HIP documentation, 2025).

Conservando la misma jerarquía de ejecución de CUDA, el lenguaje HIP conserva una estructura original del código de su homólogo, ya sea por medio de una traducción basada en reglas de sustitución de texto por medio de Hipify-Perl o por un análisis sintáctico y semántico sobre el código fuente con la herramienta Hipify-Clang.

Figura 2.

API de HIP. (HIP documentation, 2025).



ROCm proporciona bibliotecas de intermediación como *hipFFT* o *hipBLAS* que actúan como una capa de programación ligera sobre CUDA o ROCm para habilitar la compatibilidad con cualquiera de los backends. Estas bibliotecas ofrecen interfaces de memoria basadas en punteros y se integran fácilmente en las aplicaciones. HIP no está pensado como un reemplazo directo ("drop-in") de CUDA, se debe prever cierto trabajo manual de codificación y ajuste de rendimiento para GPUs AMD al portar proyectos existentes. (HIP documentation, 2025).

2.10. HIPIFY:

HIPIFY es una herramienta de ROCm que le permite a los desarrolladores a migrar la programación en GPU desde el lenguaje CUDA de Nvidia al lenguaje de programación HIP C++ de AMD, para su uso en GPU de AMD (HIPIFY documentation, s.f.). HIPIFY incluye dos herramientas que ofrecen diferentes niveles de capacidad:

2.10.1. Hipify-Clang:

Hipify-clang es una herramienta basada en *Clang* que analiza el código CUDA y lo convierte en código HIP. Traduce el código fuente en un Árbol de Sintaxis Abstracta (Abstract Syntax Tree – AST), el cual es recorrido por transformadores (matchers). Después de aplicar todos los matchers, se produce como resultado el código fuente HIP. Como ventajas, hipify-clang soporta opciones de *Clang* como -I, -D, y --cuda-path.

El soporte para nuevas versiones de CUDA es transparente, dado que el front-end de Clang está vinculado de manera estática en hipify-clang y realiza todo el análisis sintáctico de un código fuente CUDA para HIPIFY. Está muy bien soportado como una extensión del compilador.

Sin embargo, es necesario garantizar que el código CUDA de entrada sea correcto, esto debido a que un código incorrecto o con errores no puede ser traducido a HIP. Se requiere la instalación de CUDA y especificar la versión requerida mediante la opción --cuda-path. Además se deben proporcionar todos los archivos *include* y definiciones para traducir correctamente el código. (HIPIFY documentation, s.f.).

2.10.2. Hipify-Perl:

Hipify-perl es una herramienta mucho más simple, es un script basado en *Perl* que hace un uso intensivo de expresiones regulares y que se genera automáticamente a partir de hipify-clang. Reemplaza las llamadas a la API de CUDA con sus equivalentes en HIP para necesidades básicas de traducción de código útil para programas simples en CUDA. Como ventajas tiene una mayor facilidad de uso, no requiere verificaciones de corrección en el código fuente CUDA de Nvidia de entrada, no tiene dependencia de herramientas de terceros, incluida CUDA.

No obstante, presenta incapacidad o dificultad para implementar constructos como expansión de macros, namespaces, uso de directivas, templates, diferenciación entre llamadas a funciones de device o host, inserción correcta de archivos de cabecera (header files). Hipify-perl es, pero ofrece una menor detección de errores cuando surgen problemas durante la traducción. (HIPIFY documentation, s.f.).

HIPIFY puede convertir automáticamente muchas llamadas a la API de runtime de CUDA, la sintaxis de lanzamiento de kernels, funciones estándar de bibliotecas CUDA cuando existe un equivalente en HIP, y palabras clave específicas como *global* y *device*. Sin embargo,

HIP no constituye un reemplazo completo de CUDA. Después de migrar el código traducido, se debe realizar una revisión general para garantizar una traducción funcional.

2.11. Portabilidad de Implementaciones de Métodos SPH:

La portabilidad en el ámbito de la computación científica se refiere a la capacidad de una implementación de software, para ejecutarse de manera eficiente en diferentes entornos de programación sin necesidad de modificaciones significativas en el código fuente. En el contexto de los métodos SPH, la portabilidad implica la capacidad de ejecutar las simulaciones en diferentes sistemas CPU/GPU con un rendimiento comparable entre todos, dado que se trata de algoritmos muy demandantes en términos de cómputo y memoria. La falta de portabilidad implica asumir riesgos que retrasan la innovación, limitando la flexibilidad y escalabilidad de las aplicaciones. (Dongarra et al., 2017).

La portabilidad de implementaciones de métodos (SPH) en diversas plataformas, es un tema de investigación que está presente en diversos campos como la ingeniería, la física computacional y la simulación de fluidos. La capacidad para poder transferir estas implementaciones entre diferentes entornos computacionales tiene un impacto directo en la flexibilidad y la escalabilidad de las aplicaciones de simulación. (Kerscher, 2022).

La traducción de programas CUDA a HIP surge como una estrategia para abordar los desafíos de portabilidad y rendimiento en plataformas AMD. En la actualidad (como se mencionó previamente) se cuentan con herramientas HIPIFY de ROCm, que han facilitado la traducción automática de código CUDA a HIP. Algunos ejemplos conocidos incluyen el "HIPification" de un benchmark de SPH, realizado por el Barcelona Supercomputing Center (BSC por sus siglas en inglés), que obtuvieron como resultado una buena portabilidad y

reducción del tiempo de ejecución en algunas plataformas AMD. (BSC, 2024). Además, en el departamento de ciencias de la computación de la Universidad de Múnich se han realizado traducciones exitosas evaluando la portabilidad del rendimiento de HIP. (Kerscher, 2022).

A pesar de los avances que se han logrado, persisten importantes desafíos en la traducción de las implementaciones de métodos SPH, además que la existencia de diversas arquitecturas de hardware, entornos de programación e incompatibilidades son obstáculos para lograr una portabilidad óptima. Hace parte del trabajo futuro abordar estas limitaciones y desarrollar estrategias para optimizar la portabilidad entre plataformas con la menor pérdida de rendimiento posible.

3. Metodología

El análisis de portabilidad de las implementaciones de métodos numéricos SPH se abordó mediante una estrategia basada en la metodología de prototipado iterativa. Este enfoque permitió validar hipótesis en etapas sucesivas, desde la traducción inicial de los kernels CUDA hasta los intentos de integración completa bajo ROCm/HIP. Cada iteración funcionó como un prototipo parcial que ofreció retroalimentación directa sobre los problemas encontrados, facilitando ajustes progresivos en el proceso de reingeniería.

La primera fase correspondió a la investigación y recopilación de información, que incluyó la revisión de literatura especializada, la consulta de documentación oficial de CUDA y ROCm, así como el análisis de casos académicos de portabilidad en proyectos similares. En paralelo, se analizó la arquitectura de DualSPhysics con el propósito de comprender la organización del código y su grado de dependencia de CUDA.

Posteriormente, se hizo una revisión sistemática de la implementación CUDA en DualSPHysics, con el objetivo de identificar las características y optimizaciones críticas para su rendimiento, así como los elementos más sensibles al proceso de portabilidad hacia HIP. La siguiente etapa consistió en la traducción automatizada con *hipify-clang* de los archivos *.cu* a *.hip*. Luego, se llevó a cabo un proceso de reingeniería manual sobre la infraestructura en C++ (*.cpp* y *.h*), que representaba la mayor parte del proyecto, con el fin de resolver incompatibilidades y permitir la compilación bajo HIP/Clang.

Finalmente, se realiza un análisis de los resultados obtenidos con el fin de determinar la viabilidad y la eficiencia de la traducción de los archivos CUDA a HIP. Se realizará una comparación entre las plataformas CUDA y ROCm para identificar las ventajas y las limitaciones de cada una, permitiendo concluir sobre el uso futuro que se le puede dar a la portabilidad, en el desarrollo de implementaciones de métodos numéricos SPH.

Dado que no fue posible alcanzar la etapa de evaluación del rendimiento por la presencia de fallos en el proceso de enlace y limitaciones en librerías externas, el proyecto se detuvo en un estado de compilación avanzado ($\approx 98\%$). Ante ello, se definieron indicadores internos de seguimiento (TLC, MIB, TSI, BCP) con el fin de cuantificar el esfuerzo invertido y dimensionar objetivamente el grado de portabilidad alcanzado.

3.1. Entorno de desarrollo y compilación:

Las pruebas de compilación y análisis de portabilidad se realizaron en el clúster de cómputo de alto rendimiento SC3 de la Universidad Industrial de Santander (UIS). Las principales características del entorno al que se tuvo acceso son:

Tabla 1.

Comparativa de las plataformas de cómputo empleadas.

	HPE (CUDA)	**DELL (ROCm)**
CPU	(2) AMD EPYC 9554 64-Core Processor	(24) Intel(R) Xeon(R) CPU E5640 4-Core Processor
GPU	AMD Instinct MI210 - 64 GB	NVIDIA Tesla M2050 3GB
RAM	384 GB	94 GB
Flops Pico Teóricos	45 TFlops doble precisión	1.03 TFlops
Arquitectura	CDNA2	Fermi

Como se puede visualizar en la Tabla 1, la brecha computacional entre las dos máquinas utilizadas es muy amplia dado a la diferencia de más de 10 años en la concepción de las tecnologías. Es por ello que la ausencia de una máquina Nvidia con características similares a la Dell, limitaron el análisis de resultados finales presentados a continuación.

4. Resultados y Discusión

4.1. Arquitectura funcional en CUDA

Como se mencionó anteriormente DualSPHysics es un software de simulación numérica basado en el método Smoothed Particle Hydrodynamics (SPH), que combina código en C++ para la lógica de control y funcionalidades generales, con código en CUDA destinado a la ejecución de los *kernels* de cálculo intensivo. Con este enfoque híbrido, permite modelar

fenómenos de superficie libre en los que los métodos Eulerianos presentan limitaciones significativas. (DualSPHysics, 2024).

La estructura del proyecto sigue un esquema mixto **host–device**. El host, en C++, administra las operaciones de entrada/salida, la organización de datos y la gestión general de la simulación. Por su parte, los kernels CUDA implementan los cálculos principales del método SPH, que son construcción de búsqueda de partículas vecinas, interacción de partículas (fuerzas de presión y viscosidad) y actualización del estado (posición, velocidad y densidad de las partículas). En las denominadas versiones *full GPU*, estas etapas se ejecutan íntegramente en la GPU, minimizando las transferencias de datos entre CPU y GPU y maximizando el rendimiento global.

El proyecto también incorpora extensiones para multi-GPU y paralelismo a gran escala, con estrategias de balanceo de carga y técnicas de solapamiento entre transferencia de datos mediante *streaming* asíncrono. Estos mecanismos buscan aprovechar entornos de supercomputación con múltiples aceleradores.

4.1.1. Diseño de kernels y configuración de ejecución

En el diseño de los kernels CUDA, la unidad de procesamiento a nivel de hilo se establece de manera que cada hilo procese una partícula o un subconjunto de sus interacciones, empleando el modelo jerárquico *grid–block–thread*. Esta permite escalar la simulación a millones de partículas, manteniendo un aprovechamiento efectivo del paralelismo masivo de la GPU. (DualSPHysics, 2024).

De igual modo, se implementan estrategias de optimización de memoria, tales como el agrupamiento de accesos a memoria global y la utilización de memoria constante para parámetros globales de la simulación (por ejemplo, la densidad de referencia o los parámetros del kernel suavizador). Estas técnicas contribuyen a reducir la latencia y a mejorar la eficiencia de las operaciones de lectura dentro de los kernels. La documentación del proyecto señala además la existencia de estructuras específicas por módulo, en las que determinadas funcionalidades (por ejemplo la generación de oleaje mediante espectros) se implementan con kernels especializados, coordinados por clases host en C++.

4.1.2. Compilación y entorno de ejecución en CUDA

El proceso de compilación en CUDA se gestiona a través del *toolchain* oficial de Nvidia. El proyecto puede construirse en modo CPU o CPU+GPU, pero para la versión GPU resulta imprescindible el compilador *nvcc*, incluido en el *CUDA Toolkit*, además del compilador C++ estándar. Para la ejecución de binarios acelerados únicamente se requiere una GPU Nvidia con controladores actualizados.

4.1.3. Rendimiento observado

Los resultados reportados por la comunidad de usuarios y desarrolladores de DualSPHysics destacan una paralelización bastante efectiva. Se han documentado simulaciones de hasta 80 millones de partículas en GPU, con desempeños que superan ampliamente a implementaciones equivalentes en CPU multinúcleo (DualSPHysics, 2024).

En configuraciones multi-GPU, los usuarios reportan eficiencias cercanas al 100% en escalado débil, trabajando con aproximadamente 4 millones de partículas por GPU en sistemas con hasta 128 GPUs (Tesla M2090). Estos resultados se logran gracias a la reducción de costes de comunicación y al solapamiento de transferencias de datos con operaciones de cómputo (DualSPHysics, 2024).

En cuanto a los ámbitos de aplicación, DualSPHysics se ha consolidado como herramienta de referencia en la simulación de fenómenos de superficie libre, tales como la rotura de diques u olas impactando contra estructuras costeras, escenarios en los que el uso de CUDA resulta fundamental para manejar mallas de partículas de gran escala (DualSPHysics, 2024).

Este capítulo documenta el desarrollo de una versión adaptada de DualSPHysics para ROCm (AMD) mediante el uso de HIP. Se describen las decisiones técnicas, el flujo de trabajo de traducción con *hipify-clang* y el volumen de reingeniería manual necesario para integrar y compilar el proyecto bajo el *toolchain* HIP/Clang.

4.2. Estrategia de adaptación

4.2.1. Enfoque general

El proceso de adaptación se inició con el empleo de la herramienta *hipify-clang***,** aplicada exclusivamente a los archivos que contenían los *kernels* CUDA, originalmente con extensión .cu. Como resultado, dichos archivos fueron convertidos a la extensión *.hip*, lo que permitió su compilación bajo el entorno ROCm.

No obstante, este procedimiento cubrió únicamente una fracción del proyecto. La mayor parte de la infraestructura del código, implementada en archivos C++ (*.cpp* y *.h*), permaneció sin modificaciones tras la traducción automática. Estos componentes requieren un proceso de reingeniería manual, que abarcó una reescritura parcial, una reorganización estructural y resolución de incompatibilidades para garantizar la correcta compilación con Clang/HIP.

Paralelamente, se llevaron a cabo ajustes en el sistema de construcción. Fue necesario actualizar la configuración de CMake para declarar explícitamente HIP como un lenguaje de compilación nativo, marcar cada archivo *.hip* con la propiedad LANGUAGE HIP y establecer el enlace directo contra hip::device. Para realizar estas modificaciones fué necesario un enfoque manual, en contraste con el carácter automático de la traducción de *kernels*.

4.2.2. Composición del código

El análisis cuantitativo de la base de código permitió dimensionar el alcance del proceso de adaptación. El proyecto estaba compuesto inicialmente por:

- 13 archivos *.hip*, con un promedio de 711 líneas cada uno, sumando en total aproximadamente **9.243 líneas de código**.
- 90 archivos *.cpp*, con un promedio de 510 líneas, lo que equivaldría a unas **45.900 líneas**.
- 140 archivos *.h*, con un promedio de 177 líneas, alcanzando un total de **24.780 líneas**.

Tabla 2.

Composición del código del proyecto.

Tipo de Archivo	Número de Archivos	Promedio de lineas de codigo por archivo	Promedio total de líneas de código
HIP (.hip)	13	711	9243
C++ (.cpp)	90	510	45900
Headers (.h)	140	177	24780
TOTAL	243	-	79923

En conjunto, el proyecto acumulaba cerca de 80.000 líneas de código distribuidas en 243 archivos. Este desglose reveló un aspecto crítico: los archivos de *kernels* GPU, que podían ser traducidos automáticamente mediante hipify, representaban únicamente el **11,6%** del total de archivos del proyecto. En contraste, el **88,4%** restante correspondía a la infraestructura en C++, cuya integración demandó un esfuerzo manual considerable, afirmando de esta manera que la traducción automática cubre solo una parte limitada de un proyecto de gran escala.

4.2.3. Proceso técnico: Hipificación de kernels

El proceso de hipificación de kernels constituyó la primera etapa de la adaptación del código de DualSPhysics a la plataforma ROCm. En esta fase, se utilizó la herramienta hipify-clang, que como se mencionó anteriormente, fué diseñada para traducir automáticamente construcciones propias de CUDA hacia su equivalente en HIP.

En términos prácticos, la herramienta realizó una serie de transformaciones elementales pero fundamentales del proyecto. En primer lugar, se modificaron las extensiones de los archivos fuente, que pasaron de .cu a *.hip*, permitiendo que el compilador de ROCm los interpretara como archivos de GPU válidos. Posteriormente, se reemplazaron las llamadas a la API de CUDA por sus equivalentes en HIP. Entre los cambios más relevantes se encuentran el *cudaMemcpy* convertido en *hipMemcpy*, y el *cudaStream_t* sustituido por *hipStream_t*.

Otro aspecto relevante fue la traducción de la macro de lanzamiento de kernels. En CUDA, la sintaxis típica es <<<grid, block>>>, mientras que en HIP ésta debe expresarse mediante *hipLaunchKernelGGL(...)*. La conversión de este patrón fue resuelta automáticamente por la herramienta, manteniendo intacta la lógica de ejecución.

Este conjunto de transformaciones permitió preservar la estructura básica de los kernels y en consecuencia, facilitar que el código se compilara en el nuevo entorno. Sin embargo, es importante resaltar que la hipificación resolvió únicamente los elementos superficiales de la traducción, es decir la traducción directa de las llamadas y macros. Los aspectos más complejos, como la gestión de constantes de dispositivo, la integración con librerías externas y los ajustes de rendimiento específicos de la arquitectura AMD, quedaron fuera del alcance de la traducción automática y exigieron un proceso posterior de reingeniería manual.

En síntesis, la hipificación de kernels representó un primer paso exitoso y un punto de partida necesario pero insuficiente por sí mismo para garantizar la portabilidad completa del proyecto. Constituyó una fase preliminar en la que se automatizaron los cambios más evidentes, reduciendo el volumen de trabajo inicial y dejando en evidencia las limitaciones de las

herramientas de traducción automática ,cuando se enfrentan a un proyecto de gran escala y de alta complejidad como DualSPHysics.

4.2.4. Reingeniería en *.h* y *.cpp*

Concluida la fase de hipificación de los kernels, el proceso de adaptación demandó un trabajo considerable de reingeniería manual aplicado a los archivos de cabecera (*.h*) y en los módulos de implementación en C++ (*.cpp*). Esta fase constituyó un punto de inflexión, dado que gran parte de la infraestructura del proyecto DualSPHysics se encuentra en dichos archivos y la herramienta *hipify-clang* no cubre estas transformaciones.

En primer lugar, se identificaron dificultades relacionadas con las **c**onstantes de dispositivo *(__constant__)*. En CUDA, el compilador gestiona de manera automática la visibilidad de estas variables a través del *device linker*, lo que simplifica su uso en múltiples unidades de compilación. Sin embargo, en HIP/Clang este mecanismo no existe, lo que obligó a declarar explícitamente las constantes como *extern __constant__* en cada archivo donde fueran utilizadas, para así garantizar que existiera una única definición en un archivo *.hip*. Adicionalmente, la carga de valores debió realizarse mediante la función *hipMemcpyToSymbol*, un paso adicional que en CUDA no era necesario.

En segundo lugar, se detectaron problemas vinculados a las **c**abeceras y constantes matemáticas. CUDA dispone de archivos específicos como *math_constants.h*, los cuales no cuentan con equivalentes en HIP. Ante esta ausencia, se recurrió a bibliotecas estándar de C++ como <*cmath*> y <*limits*> para sustituir las referencias originales. Por ejemplo, macros como

CUDART_INF debieron reemplazarse por expresiones equivalentes como *std::numeric_limits<float>::infinity()*.

Por otro lado, un aspecto importante de reingeniería estuvo asociado a los wrappers y espacios de nombres propios del entorno CUDA. El código original incluía funciones auxiliares específicas, como *fcuda::Check_CudaErrorFun*, que no tienen un análogo en HIP. Para solventar esta carencia, se implementaron *helpers* equivalentes utilizando las funciones nativas de HIP, tales como *hipGetLastError* y *hipGetErrorString*. Con estas modificaciones se logró mantener la verificación de errores, pero bajo un enfoque independiente del *backend*.

Finalmente, fue necesario realizar modificaciones sustanciales en el sistema de compilación mediante CMake. Se introdujo la directiva *project(... LANGUAGES CXX HIP)* para declarar explícitamente el uso de HIP como lenguaje de compilación. Asimismo, se configuró cada archivo *.hip* con la propiedad *LANGUAGE HIP* y se incorporó la dependencia *hip::device* en la fase de enlazado. Adicionalmente, el orden de compilación y las rutas de las librerías externas requirieron ajustes a mayor detalle, dado que la gestión automática que CUDA ofrecía mediante macros como *cuda_add_executable* no se encuentra disponible en HIP.

En conjunto, estas acciones reflejan que la adaptación en archivos *.h* y *.cpp* no es un proceso trivial, sino una etapa que demanda un análisis detallado y soluciones específicas para cada incompatibilidad. Este esfuerzo fue determinante para alcanzar un avance significativo en la compilación del proyecto, aunque puso en evidencia la limitada cobertura de las herramientas de traducción automática y la necesidad de una reingeniería a profundidad en proyectos de gran envergadura.

Tras el trabajo de adaptación se alcanzó un **98% del build** en HIP/ROCm. En consecuencia, no se obtuvo un binario ejecutable bajo ROCm. Sin embargo, el estado avanzado de compilación y la documentación detallada de los errores encontrados permiten identificar con claridad los bloqueos técnicos y cuantificar la magnitud de la reingeniería necesaria. Al no disponer de un ejecutable funcional, no fue posible realizar comparaciones empíricas directas de tiempo de ejecución, eficiencia de GPU o escalabilidad de DualSPhysics entre CUDA y ROCm.

Esta limitación se reconoce como un resultado negativo del trabajo, pero no invalida el análisis: sirve como evidencia de los retos actuales de portabilidad práctica en proyectos HPC complejos. Para suplir la ausencia de pruebas experimentales, se presentan **r**esultados de estudios recientes que sí lograron portar aplicaciones científicas de CUDA a HIP/ROCm y compararon rendimientos.

Kuznetsov & Stegailov (2019) documentan la portación de algoritmos de **d**inámica molecular a HIP/ROCm. Su análisis muestra que, tras aplicar modificaciones manuales y un ajuste cuidadoso de parámetros de compilación, el código puede ejecutarse de forma eficiente en GPUs AMD. Los resultados indican que los tiempos de ejecución fueron comparables a los obtenidos en CUDA sobre GPUs NVIDIA, aunque el proceso requirió un ajuste especializado en la gestión de memoria y en el rendimiento de cómputo para lograr la adaptación a la arquitectura AMD. Este es un caso de importancia dada a que evidencia que la portabilidad de rendimiento es posible pero no automática, exigiendo optimizaciones adicionales más allá de la simple traducción con *hipify*, algo que es concordante con los hallazgos del presente proyecto.

Torres & Salinas (2021) desarrollaron una implementación del método de Lattice Boltzmann en ROCm como trabajo académico. El proyecto es comparable al presente trabajo

porque también aborda un método numérico complejo originalmente implementado en CUDA y enfrenta los mismos retos de compatibilidad y rendimiento. Su experiencia demostró que aunque la traducción con HIPIFY cubrió la parte sintáctica de los kernels, el mayor esfuerzo se concentró en la reingeniería manual de la infraestructura y en la resolución de problemas de linking **y** librerías externas. Además, presentaron resultados de rendimiento en GPU AMD mostrando ejecuciones competitivas con respecto a CUDA, confirmando que la dificultad no está en la capacidad del hardware, sino en el proceso de adaptación del software.

Kerscher (2022) analizó de manera sistemática la portabilidad y el rendimiento del modelo HIP en aplicaciones HPC dentro de un seminario de la GWDG. Su estudio incluyó benchmarks comparativos CUDA vs HIP en diferentes arquitecturas y concluyó que HIP logra un rendimiento prácticamente idéntico al de CUDA cuando se ejecuta sobre hardware NVIDIA, y que en hardware AMD las diferencias dependen del grado de optimización aplicado.

Durante el proceso de adaptación de DualSPHysics a ROCm/HIP se identificaron múltiples desafíos técnicos que explican por qué, a pesar de alcanzar un 98% del *build*, no se logró obtener un binario ejecutable. Estas dificultades se agrupan en categorías relacionadas con sintaxis, linking, constantes de dispositivo, orden de compilación, librerías externas y se contrastan con el comportamiento original en CUDA.

4.3. Portabilidad y Comparativa CUDA - ROCm en Modelos SPH

A fin de evaluar la portabilidad funcional y física del método SPH, se implementaron tres modelos simplificados —Poiseuille, Fuerzas Externas y Dambreak— adaptados al entorno HIP. Cada modelo permitió verificar las diferentes componentes del método, flujo laminar, aplicación de fuerzas y dinámica libre-superficie respectivamente.

4.3.1. Modelo de Poiseuille

El flujo de Poiseuille es una solución analítica clásica de la mecánica de fluidos que describe el comportamiento de un fluido viscoso en régimen laminar, confinado entre dos superficies paralelas o en el interior de un conducto. Se caracteriza por un balance entre las fuerzas viscosas y la presión impuesta en la dirección del flujo, lo que da lugar a un perfil de velocidades parabólico.

Figura 3.

Ruta jerárquica del modelo de Poiseuille.(Repositorio del proyecto, 2025)

```
poiseuille_mvp_cu
├─ src/
│  └─ poiseuille.cu
├─ include/
│  ├─ sph_kernels.hpp
│  ├─ params.hpp
│  └─ timer.hpp
├─ CMakeLists.txt
├─ bench_mvp.sbatch
└─ README.md
```

```
poiseuille_mvp_rocm
├─ src/
│  └─ poiseuille.hip
├─ include/
│  ├─ sph_kernels.hpp
│  ├─ params.hpp
│  └─ timer.hpp
├─ CMakeLists.txt
├─ bench_mvp.sbatch
└─ README.md
```

El modelo se migró desde su versión CUDA (`poiseuille.cu`) se transformó con hipify-clang y se ajustó en archivos .hpp (normalización de tipos, includes y namespaces) para compilar con Clang/HIP. Se evaluaron tamaños de bloque 128–512 threads y se fijó duration (ms) como métrica primaria de rendimiento.

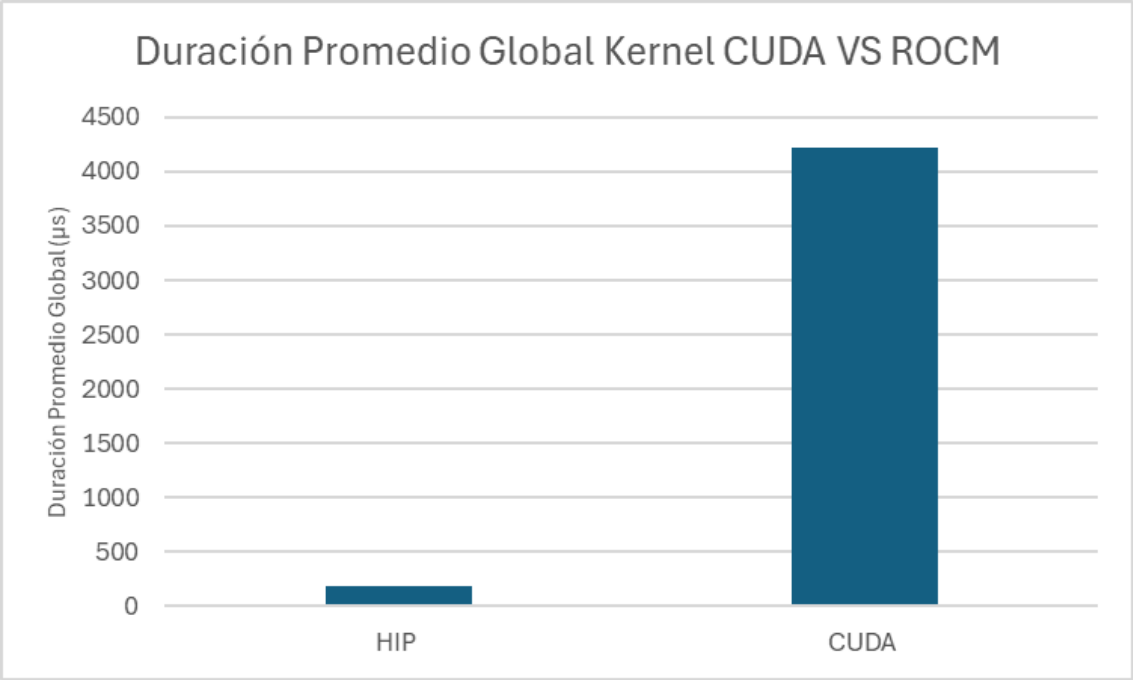
Tabla 3.

Ajustes realizados del modelo Poiseuille.

Componente	Conflicto	Causa	Solución aplicada	Observaciones
Compilación HIP	Fallos por flags y linking manual	`hipcc` requería `--offload-arch=gfx90a` y la librería `amdhip64`	Se añadió `<HIP_HIPCC_FLAGS>` y `target_link_libraries(... hip::amdhip64)`	Permite compilación directa en arquitectura MI210.
Constantes matemáticas	`math_constants.h` no disponible en ROCm	Dependencia CUDA específica	Se eliminó y reemplazó por `<cmath>` (`M_PI, std::sqrt, etc.`)	Se definen las constantes manualmente.
Tipos de datos	`typedef unsigned char byte` sentencia ambigua	Colisión con `std::byte` (C++17)	Sustituido por `std::uint8_t`	Evita conflicto con el estándar moderno C++
Kernel principal (`poiseuille.hip`)	Divergencia de tipos entre `hipStream_t` y `void*`	Declaración inconsistente en prototipo	Se unificaron prototipos: `hipStream_t stream = 0`	Corrige enlace y llamadas desde el host.

Figura 4.

Comparativo entre Kernels para el modelo Poiseuille. (Repositorio del proyecto, 2025)



El modelo de Poiseuille confirmó la portabilidad y estabilidad de los kernels: las ejecuciones fueron deterministas y el tiempo promedio por kernel (Duration, µs) resultó sensiblemente menor en ROCm/HIP que en CUDA, como se muestra en la figura 4. Este comportamiento era esperable: Poiseuille tiene acceso a memoria regular y coalescente, con *compute intensity* moderada, lo que permite que una GPU moderna bajo HIP (CDNA2) ejecute el cómputo con alta ocupación y poco overhead.

La diferencia frente a CUDA se ve amplificada por el desfase de hardware/toolchain (CUDA 8.0 en una GPU Fermi antigua), el caso base es correcto y reproduce el perfil parabólico previsto, y sirve como línea base de portabilidad; además, explica por qué en Poiseuille la brecha

es menor que en modelos SPH complejos (DamBreak), donde el patrón de acceso irregular y la presión de memoria potencian aún más la ventaja de la plataforma ROCm.

Tabla 4.

Indicadores de seguimiento para el modelo Poiseuille.

MÉTRICA	FÓRMULA	VALOR	RESULTADOS
Cobertura de Traducción Automática (TLC)	$TLC = \frac{SLOC\ traducible}{SLOC\ total} \times 100$	82,54%	El **82,54%** del código fue traducido automáticamente (archivos .cu → .hip)
Carga de integración Manual (MIB)	$MIB = 100 - TLC$	17.46%	El **17.46%** del código (archivos .cpp y .h) requirió adaptación manual.
Índice de Déficit de Traducción (TSI)	$TSI = \frac{SLOC\ no\ traducible}{SLOC\ traducicble}$	0.21	Por cada línea de código, fué necesario ajustar manualmente un aproximado de 0.20 líneas.
Progreso de compilación (BCP)	$BCP = \frac{Targets\ compilados}{Targets\ totales} \times 100$	100%	El proyecto alcanzó un **100%** del total del build.

Respecto al análisis de portabilidad de este modelo no se requirió un nivel de re ingeniería tan alto dado que en ese 17.46% de carga de integración manual la mayoría del código se soportaba correctamente en ambas plataformas CUDA/ROCM los cambios manuales fueron mínimos.

4.3.2. Modelo de Fuerzas Externas

El modelo External Forces es un componente sencillo del simulador que aplica fuerzas "externas" iguales para todas las partículas, como la gravedad. En cada paso de tiempo, toma la velocidad de cada partícula y le suma el efecto de esa fuerza, para luego actualizar su posición. No calcula interacciones entre partículas ni nada complejo: solo empuja el fluido de forma uniforme según la fuerza indicada. Por eso es rápido, fácil de comprobar y muy útil para validar que el sistema avance en el tiempo como se espera.

Figura 5.

Ruta jerárquica del modelo External Forces (Repositorio del proyecto, 2025)

```
externalForces_cu
├─ src/
|  └─ external_forces.cu
├─ include/
|  └─ sph_utils.hpp
├─ CMakeLists.txt
├─ bench_hip_prof.sbatch
└─ README.md
```

```
externalForces_hip
├─ src/
|  └─ external_forces.hip
├─ include/
|  └─ sph_utils.hpp
├─ CMakeLists.txt
├─ bench_hip_prof.sbatch
└─ README.md
```

Tabla 5.

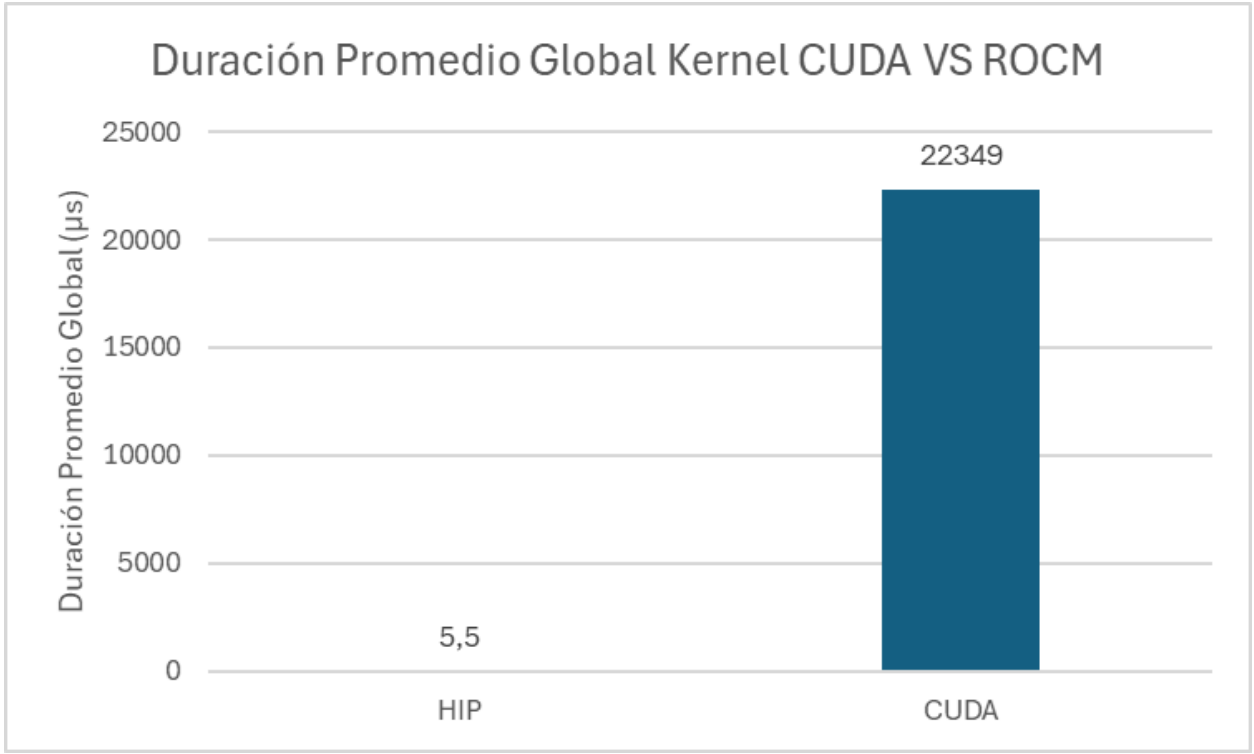
Ajustes realizados del modelo External Forces.

Componente	Conflicto	Causa	Solución aplicada	Observaciones

`External_Forces.hip`	Error de símbolos indefinidos en fase de enlace (`undefined reference to cusphinout::...`)	Las funciones dentro del namespace cusphinout no se estaban compilando o vinculando correctamente con los objetos GPU.	Se reorganizó la inclusión del namespace en los cabeceros y se actualizó el orden de enlace en CMake para compilar los HIP.	El problema surgía por diferencias en la lectura de los archivos .hip frente a .cpp dentro del flujo de compilación de Clang/HIP.
`External_Forces.hip`	Conflicto de namespace y tipos cruzados CPU/GPU	`Clang exige la definición explícita del namespace, a diferencia de nvcc.`	Se declararon explícitamente los namespaces y se unificaron las cabeceras dependientes.	Este ajuste permitió estabilizar la comunicación entre los módulos GPU y CPU.

Figura 6.

Comparativo entre Kernels para el modelo External Forces.



En el modelo *External Forces*, la portabilidad fue exitosa y el comportamiento fue consistente en ambas plataformas: el mismo kernel y la misma lógica producen resultados correctos. En rendimiento, los perfiles muestran una diferencia marcada en duración promedio (≈ 5,5 µs en ROCm/HIP vs 22 349 µs en CUDA), lo que en este entorno se traduce en una ejecución mucho más rápida en ROCm.

Esta brecha responde tanto al hardware disponible (GPU moderna en ROCm frente a una generación antigua en CUDA) como a que el kernel es simple y dominado por memoria, favoreciendo arquitecturas con mayor ancho de banda. En síntesis, el módulo se porta sin fricciones, mantiene la exactitud, y en nuestra plataforma ofrece una ventaja clara de tiempo en ROCm, dejando una base sólida para integrar fuerzas externas en escenarios SPH más complejos.

Tabla 6.

Indicadores de seguimiento para el modelo External Forces.

MÉTRICA	FÓRMULA	VALOR	RESULTADOS
Cobertura de Traducción Automática (TLC)	$TLC = \frac{SLOC\ traducible}{SLOC\ total} \times 100$	84.53	El **84.53%** del código fue traducido automáticamente (archivos .cu → .hip)
Carga de integración Manual (MIB)	$MIB = 100 - TLC$	15.46	El **15.46%** del código (archivos .cpp y .h) requirió adaptación manual.
Índice de Déficit de Traducción		1.08	Por cada línea de código, fué necesario ajustar

(TSI)	$TSI = \frac{SLOC\ no\ traducible}{SLOC\ traducicble}$		manualmente un aproximado de 1 líneas.
Progreso de compilación (BCP)	$BCP = \frac{Targets\ compilados}{Targets\ totales} \times 100$	100	El proyecto alcanzó un **100%** del total del build.

4.3.3. Modelo Dambreak

El modelo Dambreak o rotura de presa es uno de los casos de estudio más utilizados en dinámica de fluidos y en SPH, que representa la liberación súbita de un volumen de agua inicialmente en reposo, que se mueve impulsado únicamente por la acción de la gravedad, generando un frente de ola de rápida propagación.

Figura 7.

Ruta jerárquica del modelo Dambreak

```
dambreak_cu
├─ src/
|  ├─ dambreak.cu
|  └─ main.cpp
├─ include/
|  ├─ dam_kernels.hpp
|  ├─ hip_compat_constants.h
|  ├─ params.hpp
|  └─ timer.hpp
├─ CMakeLists.txt
├─ dambreak_rocprof.sbatch
└─ README.md
```

```
dambreak_hip
├─ src/
|  ├─ dambreak.hip
|  └─ main.cpp
├─ include/
|  ├─ dam_kernels.hpp
|  ├─ hip_compat_constants.h
|  ├─ params.hpp
|  └─ timer.hpp
├─ CMakeLists.txt
├─ dambreak_rocprof.sbatch
└─ README.md
```

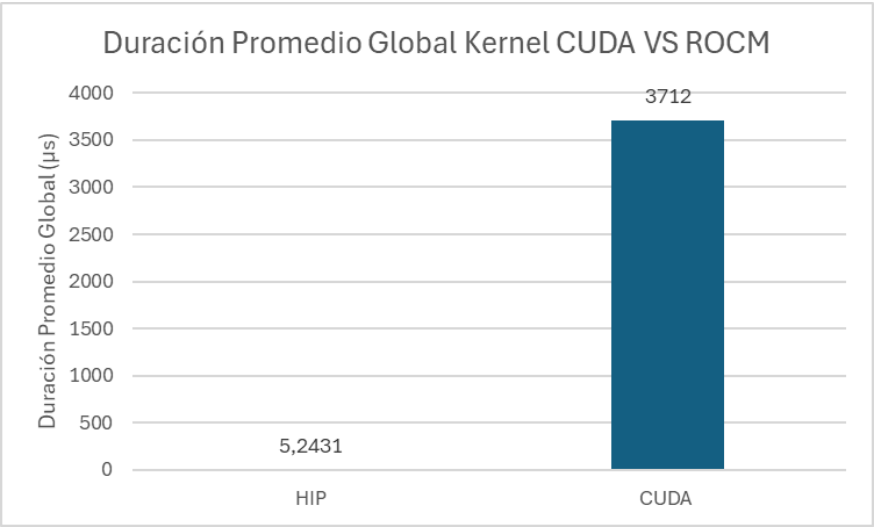
Tabla 7.

Ajustes realizados del modelo Dambreak.

Componente - Archivo	Conflicto	Causa	Solución aplicada	Observaciones
`main.cpp`	fatal error: hip/hip_runtime.h : No such file or directory	Faltaban rutas de `include ROCm`	`Se añadió ${HIP_INCLUDE_DIRS} y /opt/rocm/include`	HIP no agrega automáticamente includes al target host.
`dambreak.hip`	`undefined reference to Domain`	Namespaces dispersos	Unificación bajo `namespace dsph::dambreak`	Se centralizó definición y forward declarations.
`dambreak.hip`	`CUDART_EPSILON` indefinido	Constante de CUDA no disponible en ROCm	Reemplazo por FLT_EPSILON	Compatible con HIP y C++17.
`CMakeLists.txt`	Enlace incorrecto de librerías	Orden de linking inconsistente	Añadido `target_link_libraries(... hip::amdhip64)`	Clang requiere linking explícito con amdhip64.

Figura 8.

Comparativo entre Kernels para el modelo Dambreak.



Los resultados muestran una ventaja muy amplia de ROCm frente a CUDA en este escenario: el promedio global de duración del trazo fue ≈ 5.24 µs en ROCm y ≈ 3 712 µs en

CUDA (misma métric Duration, normalizada a μs). En términos prácticos, el kernel ejecuta varios órdenes de magnitud más rápido en la plataforma HIP/ROCm medida. Lo importante para el proyecto es que la portabilidad fue correcta (misma lógica, mismas salidas) y que, bajo esta metodología de perfilado homogénea, el tiempo por kernel favorece claramente a ROCm.

Tabla 8.

Indicadores de seguimiento del modelo Dambreak.

MÉTRICA	FÓRMULA	VALOR	RESULTADOS
Cobertura de Traducción Automática (TLC)	$TLC = \frac{SLOC\ traducible}{SLOC\ total} \times 100$	47.95	E **47,95%** del código fue traducido automáticamente (archivos .cu → .hip)
Carga de integración Manual (MIB)	$MIB = 100 - TLC$	52.04	El **52.04%** del código (archivos .cpp y .h) requirió adaptación manual.
Índice de Déficit de Traducción (TSI)	$TSI = \frac{SLOC\ no\ traducible}{SLOC\ traducicble}$	1.08	Por cada línea de código, fué necesario ajustar manualmente un aproximado de 1 líneas.
Progreso de compilación (BCP)	$BCP = \frac{Targets\ compilados}{Targets\ totales} \times 100$	100	El proyecto alcanzó un **100%** del total del build.

En el análisis de portabilidad de este modelo, la reingeniería se centró en la integración manual del flujo, porque los fallos principales estaban en firmas desalineadas y namespaces inconsistentes, con errores derivados de ambos. Tras una adaptación minuciosa (unificando

prototipos, tipos y espacios de nombres), se logró un avance de compilación del 100 % y se obtuvieron resultados correctos en ambas plataformas.

4.3.4. Modelo Dualsphysics

DualSPHysics es un simulador de dinámica de fluidos que usa el método SPH para resolver problemas con superficie libre y grandes deformaciones. Sirve para estudiar y predecir fenómenos como roturas de presa, oleaje y run-up en playas, impacto de olas en estructuras, arrastre y transporte de sólidos, interacción fluido–estructura (puentes, diques, boyas), así como flujos internos con geometrías complejas, vertidos, compuertas o canales.

Tabla 9.

Composición del código del proyecto.

Tipo de Archivo	Número de Archivos	Promedio de lineas de codigo por archivo	Promedio total de líneas de código
HIP (.hip)	13	711	9243
C++ (.cpp)	90	510	45900
Headers (.h)	140	177	24780
TOTAL	243	-	79923

La estructura del código de DualSPHysics/HIP está pensada para proyectos grandes y sostenibles. El árbol separa claramente kernels GPU (.hip) de la lógica de control en C++ (.cpp) y de las interfaces públicas (.h), lo que facilita compilar, portar y perfilar por módulos. Ese

diseño modular no es menor: el proyecto reúne 243 archivos con cerca de 80 k líneas de código, donde conviven 13 fuentes HIP (≈ 9,2 k LOC) dedicadas al cómputo masivo en GPU, 90 unidades C++ (≈ 45,9 k LOC) que orquestan memoria, condiciones de contorno y E/S, y 140 *headers* (≈ 24,8 k LOC) que encapsulan tipos, parámetros y contratos entre CPU–GPU.

Esta división por capas—API limpia en include/, implementación en src/, casos en examples/ y *tooling* de construcción con CMake—permite escalar a millones de partículas, cambiar *backends* (CUDA↔HIP), y extender funcionalidades (fuerzas externas, vecinos, integradores) sin romper dependencias. En síntesis, es un código robusto y mantenible, con responsabilidades bien aisladas y una base suficiente para crecer en rendimiento y alcance científico-industrial.

Tabla 10.

Indicadores de seguimiento Dualsphysics.

MÉTRICA	FÓRMULA	VALOR	RESULTADOS
Cobertura de Traducción Automática (TLC)	$TLC = \frac{SLOC\ traducible}{SLOC\ total} \times 100$	11.56	Solo el **11.6%** del código fue traducido automáticamente (archivos .cu → .hip)
Carga de integración Manual (MIB)	$MIB = 100 - TLC$	88.44	El **88.4%** del código (archivos .cpp y .h) requirió adaptación manual.
Índice de Déficit de Traducción		7.65	Por cada línea de código, se ajustó manualmente un

(TSI)	$TSI = \frac{SLOC\ no\ traducible}{SLOC\ traducicble}$		aproximado de 8 líneas.
Progreso de compilación (BCP)	$BCP = \frac{Targets\ compilados}{Targets\ totales} \times 100$	98.0	El proyecto alcanzó un **98%** del total del build.

La tabla evidencia que la **traducción automática** aportó poco al proyecto (**TLC = 11.56%**), por lo que la **carga de integración manual** fue muy alta (**MIB = 88.44%**). El **déficit de traducción** (**TSI = 7.65**) confirma esa exigencia: por cada línea que la herramienta pudo convertir, hubo que **ajustar ~7.6 líneas a mano** (firmas, namespaces, constantes, C++17, etc.). Aun así, el proceso fue **altamente efectivo**: se alcanzó un **98% de progreso de compilación (BCP)**, dejando al proyecto prácticamente completo y operativo.

En síntesis, aunque la portabilidad requirió un esfuerzo manual considerable, las decisiones de reingeniería fueron acertadas y permitieron **cerrar casi todo el build**; el 2% restante se perfila como trabajo puntual (librerías externas y pulido de enlaces entre módulos) más que como un bloqueo estructural.

Tabla 11.

Comparación de librerías.

LIBRERÍA	CUDA (Nvidia)	HIP (ROCm)
Chrono (Acoplar simulaciones de dinámica estructural con modelos hidrodinámicos)	Integración directa con nvcc, soporte de paralelismo GPU con CUDA kernels.	No existe implementación en ROCm, requiere reconstrucción del backend.
		Compila el código C++

MoorDyn (Interacción entre el fluido y estructuras ancladas)	Compatible a través de acoplamiento DualSPHysics–CUDA;	estándar; sin embargo, el enlace con HIP de DualSPHysics genera advertencias.
VTK (librerías para visualización, procesamiento de datos y generación gráficos 3D)	Dispone de soporte directo para la visualización de datos generados en GPU.	La versión estándar de VTK, no incluye módulos de compatibilidad HIP.
math_constants (constantes matemáticas, como π, e)	Disponible dentro del CUDA Toolkit como cabecera	ROCm carece de un equivalente directo, las constantes deben redefinirse.

El ecosistema de librerías (Chrono, MoorDyn, VTK) introduce barreras que ya exceden el alcance de un trabajo de grado. Chrono no tiene backend ROCm: implicaría diseñar/validar un motor de cómputo nuevo (APIs, kernels, pruebas de regresión). MoorDyn compila en C++ estándar, pero su enlace con DualSPHysics-HIP requiere ingeniería de integración y validación hidrodinámica cruzada. VTK carece de módulos nativos para HIP, por lo que habría que desarrollar pasarelas de memoria GPU→host/VTK y pipelines de visualización específicos. Asumir estas tareas implica meses de desarrollo, coordinación con proyectos externos, mantenimiento a largo plazo y una batería de pruebas considerable para garantizar exactitud física y estabilidad objetivos que rebasan el tiempo y los criterios académicos de un grado.

4.4. Desafíos principales

4.4.1. Tipos y espacios de nombres

Uno de los primeros problemas identificados estuvo relacionado con los tipos y espacios de nombres. En el código original existía una definición de *byte* que entraba en conflicto con *std::byte*, introducido en C++17. Mientras que en CUDA/nvcc el compilador resultó ser más permisivo y no exigía desambiguación, en HIP/Clang esta ambigüedad derivó en errores de compilación. La solución consistió en sustituir el tipo problemático por *std::uint8_t* y en algunos casos utilizar la calificación explícita *::byte*.

Tabla 12.

Desafíos técnicos.

	CUDA (Nvidia)	**HIP (ROCm)**
Estándares de lenguaje	Versión C++11	Versión (mínima) C++17
Manejo de Tipos y Namespace	El compilador NVCC es tolerante, permite ambigüedades.	El compilador Clang exige declaraciones.
Símbolos indefinidos	El compilador NVCC lo realiza a medida que se enlaza con la GPU.	Requiere la definición de *extern "C"* en cada uno de los archivos.
Unidad de ejecución	Basado en Warps de 32 hilos.	Se conforma con Wavefronts de 64 hilos.

4.4.2. Símbolos indefinidos (*undefined symbol*)

Un segundo obstáculo recurrente correspondió a los símbolos indefinidos en tiempo de enlace. Estos errores aparecían principalmente en *namespaces* como *cusphinout* o *curelaxzone* y

en funciones marcadas como *__global__*. En HIP/Clang se requiere una consistencia estricta entre declaraciones y definiciones, así como el uso de *extern "C"* cuando corresponde. En contraste, en CUDA/nvcc el *device linker* resolvía estos símbolos con mayor tolerancia. Este fué el problema crítico que bloqueó aproximadamente el 2% restante para finalizar el proceso de compilación.

4.4.3. Errores de sintaxis tolerados en *nvcc*

También se identificaron errores de sintaxis que *nvcc* toleraba. Un ejemplo fue el uso incorrecto en bucles for (por ejemplo, $for(unsigned\, p = 0;\, p(npt;\, p\, ++))$, que en CUDA lograba compilar pese a la ambigüedad, mientras que en HIP/Clang generaba errores fatales. La solución aplicada consistió en corregir explícitamente la sintaxis (por ejemplo, reemplazando por $p < npt$).

4.4.4. Constantes de dispositivo (CTE)

Otro aspecto relevante fue el manejo de las **c**onstantes de dispositivo *(__constant__)*, en el uso de la variable CTE. En HIP/Clang cada unidad de compilación debe incluir la declaración extern *__constant__* y en adición, debe existir una definición única en un archivo *.hip*. Asimismo, los valores deben copiarse utilizando *hipMemcpyToSymbol*. En CUDA/nvcc, en cambio, el *device linker* globaliza automáticamente estos símbolos, lo cual evitaba problemas de visibilidad. La ausencia de esta funcionalidad en HIP ocasionó errores como *use of undeclared identifier 'CTE'* y múltiples fallos de enlazado.

4.4.5. Linking y orden de compilación

La gestión del enlazado y el orden de compilación también supuso una dificultad considerable. En HIP fue necesario especificar manualmente propiedades como *set_source_files_properties(... LANGUAGE HIP)* y utilizar *target_link_libraries(... hip::device)*. En contraste, CUDA proporcionaba macros como *cuda_add_executable*, que resolvían automáticamente tanto el orden de compilación como el proceso de enlazado. Esta diferencia implicó tiempo adicional para la reestructuración del sistema de *build*.

4.4.6. Librerías y funciones inexistentes o divergentes

Se presentaron asimismo limitaciones relacionadas con librerías y funciones inexistentes o divergentes. Un ejemplo fue la ausencia de *hip/math_constants.h*, lo que obligó a redefinir manualmente constantes como CUDART_PI_F o CUDART_INF empleando <limits> y <cmath>. De igual forma, funciones auxiliares como *fcuda::Check_CudaErrorFun* no cuentan con equivalente directo en HIP, por lo que se implementaron wrappers propios basados en *hipGetLastError* y *hipGetErrorString*. En CUDA estas cabeceras y utilidades están disponibles en el *toolkit* oficial, lo que resalta la diferencia de madurez entre ambos ecosistemas.

4.4.7. Manejo estricto de errores en runtime HIP

En cuanto al manejo de errores en tiempo de ejecución, HIP aplica un control más estricto, en la cuál funciones como *hipMalloc*, *hipMemcpy* y *hipFree* se encuentran anotadas con *[[nodiscard]]*, lo que genera advertencias o errores de compilación si se ignoran sus valores de retorno. En CUDA/nvcc esta verificación no era obligatoria. La solución fue realizar modificaciones sistemáticas, capturando explícitamente el valor de retorno en una variable *hipError_t ,* o cuando no era necesario realizando un *cast* a (void).

4.4.8. OpenMP y librerías externas

Finalmente, se detectaron problemas relacionados con OpenMP y librerías externas. La integración de OpenMP con Clang/ROCm presentó dificultades en su detección, y se observaron fallos de compilación y enlazado con bibliotecas externas como *Chrono*, *MoorDyn* y *VTK*. Mientras que en el ecosistema CUDA+GCC estas integraciones resultaban estables y probadas, en HIP/Clang la falta de compatibilidad plena contribuyó a la imposibilidad de generar un binario ejecutable funcional.

4.4.9. Limitaciones estructurales observadas

Se evidenció una cobertura reducida de hipify, en la cuál solo procesó 13 archivos *.hip* (~11.6% del total). El 88.4% restante (headers y *.cpp*) no quedaron incluidos requiriendo ajustes de reingeniería manual. Pese a llegar al **98% de build** completado, los problemas de linking y librerías externas impidieron un binario ejecutable. Esto se debe a la ausencia de equivalentes

directos de varias cabeceras y menor madurez de las herramientas ROCm comparado con CUDA.

Tabla 13.

Modificaciones a nivel de librería C++.

Patrón heredado (C++ 11)	Reemplazo para C++ 17	Observaciones
`std::auto_ptr<T>`	`std::unique_ptr<T>()`	Propiedad exclusiva
`std::result_of<F(Args…)>`	`std::invoke_result_t<F,Args…>`	Requiere `<type_traits>` y si se invoca `<functional>`
`std::random_shuffle`	`std::shuffle(first,last,rng)`	Necesita `<random>` y un RNG tipo `std::mt19937`

`std::bind1st/bind2nd`
`std::ptr_fun`
`std::men_fun`
	`std::bind`
	o reemplazar lambdas
		Los lambdas simplifican y evitan ambigüedades de tipos.

`std::unary_function/binary_function`	Solo se elimina la función	Estructuras base ya no se utilizan.
`std::iterator`	Se deben definir *traits* manuales	Se puede usar iteradores modernos/ranges.

4.5. Recomendaciones

Estas recomendaciones proponen una hoja de ruta práctica para abordar proyectos de portabilidad de cómputo entre plataformas (CUDA→HIP/ROCm), orientada a equipos que necesitan avanzar poco riesgo, medición objetiva y resultados reproducibles. La metodología es incremental: comienza con un modelo viable que valide la viabilidad técnica, escala por módulos, la optimización y comparación de rendimiento con una métrica única de tiempo. En

otras palabras, es una guía operativa que prioriza claridad, control y reproducibilidad, adaptable al tamaño y restricciones de cada proyecto.

Fase 0. Preparación: Esta fase inicial tiene como objetivo conocer qué es lo que se debe portar del proyecto y con qué se cuenta.

- Inventario de código: se deben tomar información referente a kernels, librerías externas, scripts de build (CMake/Makefile), pruebas y datos, del proyecto que se está abordando.
- Establecer un entorno reproducible, con versiones de compiladores, toolchains, drivers, SO, GPU/CPU.
- Compilar y perfilar el proyecto original para guardar tiempos, consumo y exactitud.
- Métricas de seguimiento como TLC, MIB, TSI, BCP (ver definiciones abajo).

Fase 1. Prototipo mínimo viable (MVP): Este primer paso busca validar la viabilidad sin migrar todo el proyecto.

- Se debe escoger un módulo representativo y pequeño (kernel simple).
- Se traduce con la herramienta disponible (p. ej., `hipify`) y se revisa manualmente.
- Es necesario realizar la sustitución de cabeceras específicas del origen por equivalentes estándar (p. ej., `math_constants.h` → `<cmath>/<cfloat>`).
- Normalizar los tipos y namespaces, evitar el uso de `byte`, reemplazarlo por `std::uint8_t`. También se recomienda usar un solo namespace coherente.
- Unificar firmas host/device (streams, punteros, const-correctness).

- Establecer wrappers de error neutros, una macro/función que funcione igual en ambos backends.
- Realizar un build limpio, separando flags del host y del backend. No se debe mezclar flags de offload con el compilador C++.

Fase 2. Escalamiento por módulos: En esta segunda fase, se tiene como objetivo crecer del MVP a componentes con mayor complejidad. Se sugiere un orden en que se aborde: kernels sin vecindario → estructuras de datos → interacción principal → integradores/IO. Para cada módulo, se debe compilar y ejecutar un test pequeño, para registrar métricas (tiempos/uso de memoria).

Fase 3. Dependencias externas (criterio de alcance): La tercera fase busca definir qué se debe integrar y qué se debe documentar como trabajo futuro. Realizar una clasificación de librerías, nativas del backend destino, soportadas parcialmente, o sin soporte. Si no hay soporte se debe documentar la brecha (API, esfuerzo estimado, riesgos).

Fase 4. Optimización inicial (backend destino): Esta cuarta fase, tiene como objetivo lograr un rendimiento razonable antes de micro-optimizar.

- Se recomienda establecer tamaños de bloque acordes al hardware (p. ej., múltiplos de 64 en el caso de los wavefronts AMD).
- Evaluar la memoria, accesos alineados y contiguos, usar compartida cuando sea pertinente, considerar una alineación 128 bits.

- Reproducibilidad numérica: si es crítica, se debe desactivar contracciones FMA (p. ej., -ffp-contract=off) y fijar tolerancias de comparación.
- Integrar el perfilador nativo como rocprof, Nsight, VTune, entre otros, desde el inicio del proyecto.

Fase 5. Medición y comparación: La quinta fase busca comparar con una métrica homogénea.

- Es recomendable trabajar con una sola métrica de tiempo a nivel de kernel (p. ej., Duration), en la misma unidad (μs).
- Registrar el promedio global y promedio por kernel. En este punto es a criterio si se excluye memcpy para medir sólo cómputo.
- Realizar una contextualización del hardware TFLOPs, ancho de banda, para interpretar diferencias.

Fase 6. Gestión y estrategia (transversal): En la sexta fase, el foco se dirige hacia la monitorización del progreso y la minimización de las repeticiones (erradicación de reprocesos).

- Se recomienda mantener las métricas TLC/MIB/TSI/BCP por módulo.
- Usar una matriz de problemas, en el que se pueda visualizar el problema, su causa en destino y la solución aplicada.
- Versionar en ramas pequeñas, documentando cada módulo y señalando cómo compilar, cómo perfilar y los resultados obtenidos.

Fase 7. Cierre: La fase final busca dejar el proyecto utilizable y auditable.

- Se debe ajustar el código con estructura clara (include/, src/, scripts).
- Implementar cripts de build y perfilado reutilizables (batch/Slurm si aplica).
- Presentar un informe con la metodología utilizada, los resultados obtenidos, los límites (dependencias sin soporte) y la línea por la cual se abordará el trabajo futuro.

5. Conclusiones y Trabajo Futuro

El análisis realizado muestra que la portabilidad de aplicaciones científicas a gran escala, como DualSPHysics escrito originalmente en CUDA, hacia ROCm (AMD) presenta limitaciones relevantes. Aunque se alcanzó un ≈98% de progreso de compilación, la presencia de símbolos indefinidos, la falta de librerías equivalentes, diferencias en el manejo de constantes y en el proceso de enlace, así como incidencias asociadas al salto de C++11 a C++17/21, impidieron generar un binario ejecutable completo. En su estado actual, herramientas automáticas como hipify-clang son valiosas para la conversión sintáctica de kernels, pero insuficientes para garantizar una portabilidad integral en proyectos con arquitecturas complejas.

Pese a ello, el proceso permitió identificar patrones de error y derivar una metodología operativa que orienta futuros esfuerzos de migración entre CUDA y HIP. Para objetivar el avance y la carga de reingeniería, se definieron indicadores internos (TLC, MIB, TSI, BCP) que facilitaron cuantificar costos técnicos y ubicar cuellos de botella en la compilación. Además, con base en nuestros propios modelos implementados y perfilados (de menor a mayor complejidad), observamos que la ejecución en ROCm/HIP alcanzó duraciones promedio de kernel sustancialmente menores que en el entorno de referencia en CUDA, manteniendo la coherencia

física de los resultados. Si bien estas diferencias están influidas por el hardware y el toolchain disponibles, la evidencia empírica respalda que la portabilidad funcional es viable y que el backend ROCm puede explotar eficientemente los kernels portados bajo una metodología de medición homogénea.

Como trabajo futuro, se recomienda un enfoque incremental: prototipar con módulos representativos de menor escala, aplicar refactorización progresiva y monitorear la evolución de ROCm en versiones posteriores. Es importante priorizar el soporte de librerías externas y el fortalecimiento del compilador HIP/Clang para consolidar un entorno de portabilidad más robusto. En una perspectiva de futuro tecnológico, la dirección natural del ecosistema HPC es unificar plataformas y software mediante capas de abstracción portables, APIs neutrales de proveedor, formatos intermedios comunes y toolchains compatibles, de modo que el mismo código fuente pueda compilarse y ejecutarse con rendimiento razonable sobre arquitecturas heterogéneas.

Avanzar hacia esa convergencia facilitará superar las limitaciones actuales y viabiliza la adopción de aplicaciones de dinámica de fluidos y simulaciones científicas complejas en múltiples backends sin rehacer el código cada vez.

Referencias Bibliográficas

Repositorio del proyecto, Análisis de la portabilidad de la implementación de métodos numéricos de hidrodinámica de partículas suaves en diferentes plataformas y frameworks CPU/GPU. 2025;10. https://github.com/stevenU19/modelos_SPH/tree/main

Long S, Wong KKL, Fan X, Guo X, Yang C. Smoothed particle hydrodynamics method for free surface flow based on MPI parallel computing. Frontiers In Physics. 2023;11. doi:10.3389/fphy.2023.1141972.https://www.frontiersin.org/journals/physics/articles/10.3389/fphy.2023.1141972/full

Yang, Q., Tan, Q., Ren, Y., Fang, H., Hu, M., & Bao, A. (2024). Smoothed Particle Hydrodynamics (SPH) Analysis of Slope Soil–Retaining Wall Interaction and Retaining Wall Motion Response. Processes, 12(2), 411. https://doi.org/10.3390/pr12020411

Wang, Z.-B., Chen, R., Wang, H., Liao, Q., Zhu, X., & Li, S.-Z. (2016). An overview of smoothed particle hydrodynamics for simulating multiphase flow. Applied Mathematical Modelling, 40(23–24), 9625–9655. https://doi.org/10.1016/j.apm.2016.06.037

Anderssén, V. (2024). Converting CUDA programs to run on AMD GPUs (Master's Thesis). Åbo Akademi University, Faculty of Science and Engineering. https://link.springer.com/chapter/10.1007/978-3-030-36592-9_11

Monaghan, Joseph. (2005). Smoothed Particle Hydrodynamics. Reports on Progress in Physics. 68.1703.10.1088/0034-4885/68/8/R01.https://www.researchgate.net/publication/230988821_Smoothed_Particle_Hydrodynamics

Krog, Øystein & Elster, Anne. (2010). Fast GPU-based fluid simulations using SPH. Applied Parallel and Scientific Computing. 7134. 98-109. 10.1007/978-3-642-28145-7_10. https://www.researchgate.net/publication/220840352_Fast_GPU-based_fluid_simulations_using_SPH

Domínguez, J. M., Crespo, A. J. C., Valdez-Balderas, D., Rogers, B. D., & Gómez-Gesteira, M. (2015, agosto). New multi-GPU implementation for smoothed particle hydrodynamics on heterogeneous clusters. Computer Physics Communications, 184(8), 1848–1860. https://doi.org/10.1016/j.cpc.2013.03.008

Grassa, J. M. (2004). El método SPH. Aplicaciones en ingeniería marítima. Revista Digital del Cedex, (133), 37–50. https://ingenieriacivil.cedex.es/index.php/ingenieria-civil/article/view/2099

Torres, C., & Salinas, Á. (2021, 1 de agosto). Implementación en AMD ROCm de una simulación computacional del método de Lattice Boltzmann [Tesis de pregrado, Universidad Técnica Federico Santa María]. Repositorio Institucional USM. https://repositorio.usm.cl/handle/11673/52569

Kerscher, N. (2022, 4 de julio). Investigating the HIP programming model with regards to portability and performance portability. Seminario sobre programación portátil de aplicaciones HPC, GWDG. https://events.gwdg.de/event/243/contributions/506/attachments/142/179/FINAL_Seminar_on__Performance__Portable_Programming_of_HPC_Applications___Topic__HiP.pdf

Pericacho Ávila, J. (2023). Integración de HIP/ROCm en un modelo de programación paralela heterogénea [Trabajo fin de máster, Universidad de Valladolid]. UVaDoc. https://uvadoc.uva.es/handle/10324/63027

Lewis, T. (2025, septiembre 15). *Specifications - OpenMP*. OpenMP. https://www.openmp.org/specifications

IBM. (s. f.). (Consultado 2025, 17 de agosto) *¿Qué es la computación de alto rendimiento (HPC)?* https://www.ibm.com/es-es/topics/hpc

Nvidia, (2025, septiembre 2). Introduction — NVIDIA CUDA Compiler Driver 13.0 documentation. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc.

AMD. (2024). HIPIFY-clang: CUDA to HIP source translator (Versión ROCm 6.1.1). ROCm Documentation.https://rocm.docs.amd.com/projects/HIPIFY/en/docs-6.1.1/hipify-clang.html

AMD. (2024). OpenMP support in ROCm LLVM project. ROCm Documentation. https://rocm.docs.amd.com/projects/llvm-project/en/latest/conceptual/openmp.html

AMD. (Consultado 2025, 15 de febrero). ROCm open software. AMD. https://www.amd.com/en/products/software/rocm.html

ROCm. (Consultado 2025, 20 de febrero). HIPIFY: Convert CUDA to portable C++ code. GitHub. https://github.com/ROCm/HIPIFY

HIP documentation. (2025, 28 de agosto). HIP 7.0.51831 Documentation. https://rocm.docs.amd.com/projects/HIP/en/latest/index.html.

HIPIFY documentation. (Consultado 2025, 18 de febrero). HIPIFY Documentation. https://rocm.docs.amd.com/projects/HIPIFY/en/latest/index.html.

DualSPHysics. (Consultado 2025, 03 de marzo). DualSPHysics: A combined CUDA and OpenMP implementation of the Smoothed Particle Hydrodynamics method based on the advanced SPHysics code. https://dual.sphysics.org/

DualSPHysics. Home. (Consultado 2025, 29 de febrero). GitHub. https://github.com/DualSPHysics/DualSPHysics/wiki.

Barcelona Supercomputing Centre (BSC-CNS). (2024, 2 de agosto). Discover BSC: The centre. https://www.bsc.es/discover-bsc/the-centre