Facultad
de
Ciencias

Análisis del impacto de la paralelización y la vectorización sobre el rendimiento en cargas de trabajo HPC

(Analysis of the impact of parallelization and vectorization on performance in HPC workloads)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Daniel Giménez Cianca

Director: Borja Pérez Pavón

Julio – 2025

Índice

Resumen

En las últimas décadas, la Ley de Moore, que predecía la duplicación del número de transistores en un chip aproximadamente cada dos años, ha sido el motor del crecimiento exponencial en la capacidad de cómputo de los procesadores. Sin embargo, este ritmo de escalado se ha ralentizado drásticamente en los últimos años debido a limitaciones físicas y económicas en la miniaturización de los transistores. Esto ha provocado un cambio de paradigma: ya no se puede confiar exclusivamente en el aumento de frecuencia o en la densidad de transistores para mejorar el rendimiento.

Ante este escenario, la industria se ha volcado hacia arquitecturas paralelas, incorporando múltiples núcleos por procesador y unidades vectoriales (SIMD), que permiten ejecutar múltiples instrucciones simultáneamente o aplicar una misma operación sobre varios datos en paralelo. Aprovechar este tipo de hardware requiere una programación consciente del paralelismo, tanto a nivel de hilos como de datos, así como un entendimiento profundo del comportamiento de las aplicaciones sobre la jerarquía de memoria.

Este trabajo de fin de grado se centra en el análisis y evaluación de distintas implementaciones paralelas de un conjunto representativo de benchmarks, haciendo uso de tecnologías como OpenMP e instrucciones SIMD, en un entorno controlado de computación de alto rendimiento. Se estudia cómo el rendimiento de una misma aplicación puede variar significativamente según el número de hilos empleados y el tipo de vectorización aplicada (manual o automática).

El estudio se ha realizado sobre nueve benchmarks pertenecientes a distintos Berkeley Dwarfs, patrones computacionales que permiten clasificar las aplicaciones según su estructura de acceso a datos y tipo de cómputo. Cada benchmark ha sido ejecutado en varias versiones y bajo diferentes configuraciones, evaluando su comportamiento con métricas obtenidas a través de Intel VTune Profiler, como el tiempo de ejecución, la eficiencia del uso de la caché, el CPI o el acceso a DRAM.

El objetivo principal es identificar qué técnicas de optimización son más efectivas según el tipo de aplicación, estableciendo una correlación entre patrón computacional, tipo de vectorización y escalabilidad. Con ello, se busca ofrecer una visión fundamentada que ayude a seleccionar estrategias de optimización adecuadas en función del tipo de carga de trabajo.

Abstract

In recent decades, Moore's Law, which predicted that the number of transistors on a chip would double approximately every two years, has been the driving force behind exponential growth in processor computing power. However, this rate of scaling has slowed dramatically in recent years due to physical and economic limitations in transistor miniaturization. This has led to a paradigm shift: it is no longer possible to rely exclusively on increasing frequency or transistor density to improve performance.

Faced with this scenario, the industry has turned to parallel architectures, incorporating multiple cores per processor and vector units (SIMD), which allow multiple instructions to be executed simultaneously or the same operation to be applied to multiple data sets in parallel. Taking advantage of this type of hardware requires programming that is conscious of parallelism, both at the thread and data levels, as well as a deep understanding of the behavior of applications on the memory hierarchy.

This final degree project focuses on the analysis and evaluation of different parallel implementations of a representative set of benchmarks, using technologies such as OpenMP and SIMD instructions, in a controlled high-performance computing environment. It studies how the performance of the same application can vary significantly depending on the number of threads used and the type of vectorization applied (manual or automatic).

The study was conducted on nine benchmarks belonging to different Berkeley Dwarfs, computational patterns that allow applications to be classified according to their data access structure and type of computation. Each benchmark was run in several versions and under different configurations, evaluating its behavior with metrics obtained through Intel VTune Profiler, such as execution time, cache usage efficiency, CPI, and DRAM access.

The main objective is to identify which optimization techniques are most effective depending on the type of application, establishing a correlation between computational pattern, type of vectorization, and scalability. The aim is to provide an informed view that helps select appropriate optimization strategies based on the type of workload.

1. Background

En este capítulo se describen las herramientas que se van a utilizar en el desarrollo de este proyecto, así como también, se exponen los conocimientos teóricos y técnicos fundamentales que sirven de base para la correcta comprensión del desarrollo y los resultados presentados. En particular, se abordarán brevemente conceptos clave relacionados con la herramienta de análisis de rendimiento Intel VTune, técnicas de optimización a nivel de aplicación como el paralelismo y la vectorización, el uso de OpenMP e instrucciones SIMD.

1.1. VTUNE

VTune es una herramienta de análisis de rendimiento para máquinas con arquitectura basada en x86 que funciona tanto en sistemas operativos Windows como Linux. Esta herramienta permite analizar el comportamiento de un sistema a nivel de hardware y software con el fin de identificar cuellos de botella, ineficiencias o patrones críticos que afecten el tiempo de ejecución. En la figura 1 se puede observar la interfaz gráfica de la herramienta.

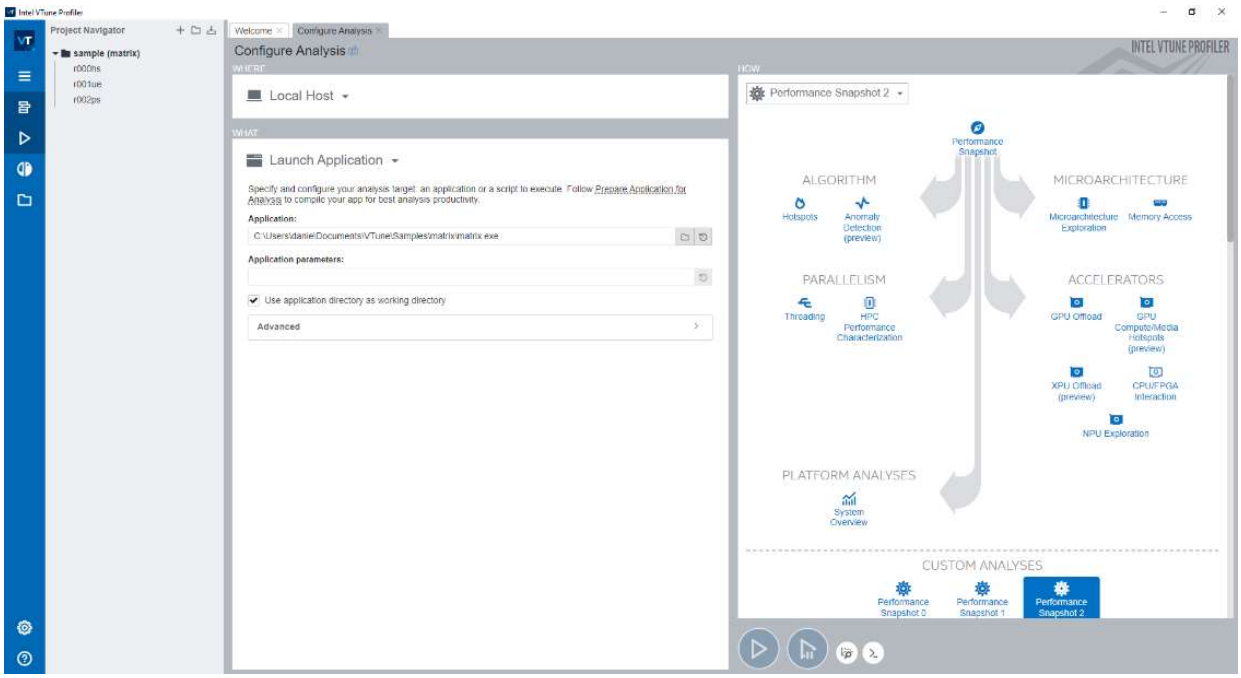

Figura 1: Interfaz gráfica de VTune.

Para la ejecución de los análisis se ha usado la versión de Linux y para el análisis de resultados se ha utilizado ambas, pero no hay diferencia a la hora de observar resultados. Para la ejecución de análisis VTune consta de 2 formas de uso:

- Una interfaz gráfica (GUI) orientada a la exploración visual y navegación interactiva de los resultados
- Y una interfaz de línea de comandos (CLI), ideal para ejecutar análisis de forma automatizada o integrarlos en flujos de evaluación más complejos.

La configuración de un análisis a través de la interfaz gráfica consta de tres partes diferenciadas. La primera, se trata de la selección de donde (WHERE) se va a realizar el análisis. Los análisis se pueden realizar en local, tanto en equipos Linux o Windows, o en remoto, donde podemos encontrar sistemas embebidos y Android. Debajo de esta misma, se encuentra la selección de qué (WHAT) ejecutar, un ejecutable, un proceso o un sistema completo. Una vez seleccionado el tipo de aplicación que se va a lanzar, tendremos que poner el path absoluto al ejecutable y los parámetros de entrada. Por último, a la derecha se encuentra la selección de cómo (HOW) queremos realizar el análisis. Haciendo clic, se abre un árbol que muestra los distintos tipos de análisis, donde son desglosados en seis grupos de análisis dependiendo del área que se quiera analizar más en profundidad, estos grupos son: algoritmos, microarquitectura, paralelismo, entrada/salida, aceleradores y plataforma. Los principales grupos con sus análisis son:

El primer análisis y más general:

- *Performance snapshot*: es un buen comienzo porque muestra donde hay más problemas y donde se deberían focalizar los siguientes análisis.

Dentro del grupo de análisis de algoritmos tenemos:

- *Hotspots*: es un tipo de análisis que ayuda a la hora de encontrar dónde dentro del código se consume la mayoría del tiempo de ejecución del sistema que estamos observando.
- *Anomaly Detection*: permite identificar anomalías en el rendimiento de ciertos intervalos que se repiten frecuentemente como pueden ser bucles iterativos.
- *Memory Consumption*: se encarga del análisis del consumo de memoria realizado por la aplicación que se está ejecutando, lo que implica, un análisis de memoria detallado sobre los objetos de memoria y cómo se asignan, pero este análisis solo está soportado por sistemas Linux.

Dentro del grupo análisis de microarquitectura:

- *Microarchitecture Exploration*: es el mejor análisis para identificar los estados del pipeline de la CPU y para encontrar que hardware es el que está causando cuellos de botella en la aplicación.
- *Memory Access*: es el mejor análisis para aplicaciones que estén limitadas por memoria, ya que muestra qué niveles de la jerarquía de memoria tienen

mayor impacto en el rendimiento de la aplicación, mostrando el uso tanto de la memoria caché como de la memoria principal.

El siguiente grupo de análisis es el de paralelismo:

- *Threading*: es el mejor para visualizar el paralelismo a nivel de thread, localizando en qué parte del código hay baja concurrencia e identificando los cuellos de botella del código de la aplicación.
- *HPC Performance Characterization*: se usa para analizar aplicaciones más intensivas en cómputo y como éstas usan los recursos como la CPU o las unidades de punto flotante.

Por último, encontramos el grupo de plataforma:

- *System Overview*: solamente es un análisis general que no usa eventos basados en drivers para su realización, lo cual sugiere que es menos profundo que los anteriores, pero que sirve para identificar a nivel más general cuáles pueden ser los limitantes de la aplicación.

Fuera de estos modos predeterminados de análisis, VTune permite la generación de análisis completamente personalizados, haciendo uso de los contadores de hardware del procesador y del sistema operativo.

Al hacer uso de procesadores Intel, VTune permite un uso poco intrusivo, que utiliza contadores hardware de Intel de una forma mucho más cómoda que otras herramientas de análisis de rendimiento. También contamos con los contadores ftraces que proporciona Linux como sistema operativo. A pesar de que VTune no es muy intrusivo, conlleva un overhead derivado de los ciclos de muestreo que realiza, lo que también implica que, dependiendo del tipo de análisis, habrá un mayor o menor overhead, este overhead se observa en la tabla 1.

Tipo de análisis	Técnica usada	Overhead
Hotspots	Muestreo (sampling)	~1–5%
Memory Access / Microarch.	Sampling + modelos	~5–15%
HPC Performance Characterization	Sampling	~5–10%
Instrumentation (tracing)	Instrumentación	~20–50%

Tabla 1: Overhead estimado según tipo de análisis en Intel VTune [1].

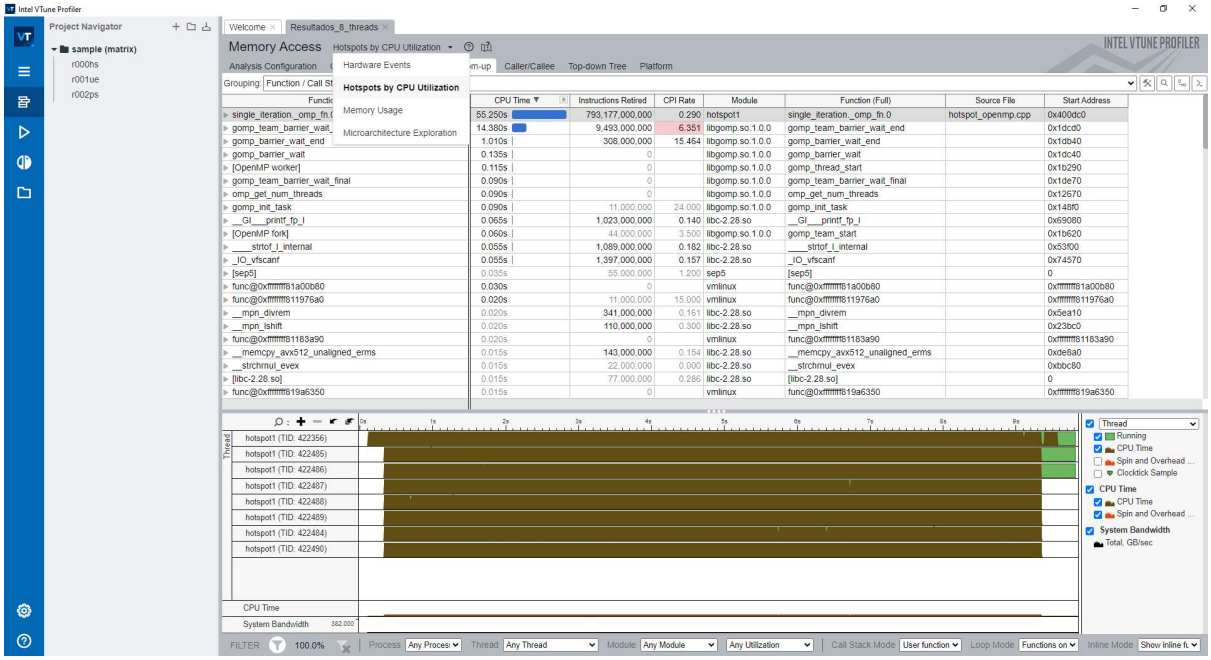

Figura 2: Desglose del análisis en VTune.

Los análisis constan de la siguiente información, como se muestra en la figura 2. La primera pestaña, muestra la configuración del análisis (Analysis Configuration). La segunda, muestra un log (Collection Log) con todos los eventos que han ocurrido durante la ejecución del análisis. La tercera, muestra un resumen general (Summary) de los resultados del análisis, donde se muestran los cuellos de botella e información superficial. La cuarta pestaña, muestra ya información detallada del análisis (Bottom-up) donde se puede desglosar dicha información y ordenar dependiendo de que quiera el usuario. También, consta de gráficas temporales de cómo ha sido la ejecución, mostrando el tiempo de CPU, el uso de memoria y otros posibles datos que quiera seleccionar el usuario. A partir de aquí, dependiendo del análisis que seleccionemos, habrá distintos tipos de formas de mostrar la información, ya sea, contadores de eventos (Event Count), desgloses en forma de árbol (Top-down Tree), etc.

VTune también provee al usuario con herramientas para generar ficheros de distintos tipos a partir de análisis de aplicaciones para facilitar la automatización de los resultados. Por ejemplo, VTune permite transformar un análisis en un archivo csv (comma-separated values), que es un formato que permite mostrar información visible desde Excel y que ayuda a la automatización de la información para su posterior análisis. Gracias a esto es posible generar gráficas a partir de la información obtenida de los análisis y con ello un mejor análisis de los resultados.

1.2. Paralelización

Para mejorar el rendimiento computacional de los sistemas actuales, se recurre a la paralelización, que consiste en dividir el trabajo en porciones independientes que puedan ejecutarse en elementos de cómputo distintos. Existen distintos tipos de paralelismo, siendo los más comunes:

- *Paralelismo a nivel de hilos (Thread-level Parallelism)*: se crean múltiples hilos de ejecución que comparten el mismo espacio de memoria y se reparten el trabajo dentro de una misma tarea.
- *Paralelismo a nivel de datos (Data Parallelism)*: una misma operación se aplica a múltiples elementos de datos al mismo tiempo.

Existen diferentes tipos de modelos de programación que implementan estos paradigmas subyacentes. Los cuatro tipos principales y más utilizados son:

- *Memoria compartida (shared-memory)*: los hilos comparten un espacio de memoria global, es decir, comparten datos. OpenMP es un ejemplo de este paradigma.
- *Memoria distribuida (distributed-memory)*: cada proceso tiene su propio espacio de memoria y se comunica con otros mediante paso de mensajes. MPI es el estándar más conocido.
- *Modelo híbrido*: combina los dos anteriores, por ejemplo, combinando MPI y OpenMP, para aprovechar arquitecturas multinodo y multinúcleo.
- *Modelo host-device*: típico de sistemas heterogéneos con CPU y aceleradores (como GPU), en el que el dispositivo anfitrión (host) gestiona la ejecución y transfiere datos al dispositivo (device), donde se ejecuta el cómputo paralelo. Este modelo es habitual en entornos como CUDA o OpenCL.

Este proyecto se centra en el enfoque en arquitecturas de memoria compartida, utilizando OpenMP para la paralelización a nivel de hilos e instrucciones SIMD para la vectorización. Por tanto, el análisis se centra principalmente al paralelismo de hilos y al paralelismo de datos en sistemas homogéneos.

1.3. OpenMP

En el ámbito de la programación paralela sobre arquitecturas de memoria compartida, existen diversos modelos y herramientas que permiten al programador explotar el paralelismo de forma eficiente. Uno de los más utilizados es OpenMP (Open Multi-Processing), una API que facilita la paralelización mediante directivas insertadas en el código fuente, permitiendo expresar el paralelismo de manera sencilla y sin

grandes modificaciones del código secuencial original [2]. Este modelo se adapta bien a sistemas multinúcleo, donde los hilos comparten el mismo espacio de memoria.

Además de OpenMP, destacan otras implementaciones orientadas también a entornos de memoria compartida. Por ejemplo, Cilk, desarrollado en el MIT, permite la creación eficiente de tareas paralelas mediante un modelo basado en la recursividad y el "work stealing" como estrategia de planificación [3]. Por otro lado, Intel Threading Building Blocks (TBB) ofrece una biblioteca de alto nivel para C++ que abstrae el paralelismo mediante estructuras de tareas y contenedores concurrentes, facilitando la escalabilidad en arquitecturas modernas.

OpenMP se basa en un modelo tipo fork-join, lo que significa que un hilo maestro se encarga de ejecutar el inicio del programa y este mismo, gracias a las directivas #pragma omp (en C y C++), crea hilos (fork) que ejecutan un bloque de código en paralelo. Al finalizar el bloque de ejecución, el hilo se vuelve a incorporar al hilo maestro (join) y se sincronizan. Por ello, OpenMP gracias a las directivas `pragma` ofrece una forma fácil de añadir paralelismo a código ya existente. OpenMP también permite ajustar dinámicamente el número de hilos que van a ser usados en la ejecución. Por último, dispone de mecanismos para el control de la concurrencia como regiones críticas, de exclusión mutua, barreras, etc.

El runtime de OpenMP es la parte de la implementación que se encarga de gestionar la ejecución paralela del programa en tiempo de ejecución. Su función principal es coordinar la creación, gestión y sincronización de los hilos. Mientras que el compilador traduce las directivas de OpenMP en llamadas al runtime, este último es el responsable de controlar cómo se lleva a cabo realmente la paralelización durante la ejecución del programa.

OpenMP se basa en su mayoría en el uso de directivas, que son anotaciones en el código que le dicen al compilador qué secciones del código han de ser ejecutadas en paralelo, y las traduce en llamadas al runtime. Algunas de estas directivas y las más relevantes en el proyecto son:

- *`#pragma omp parallel`*: Muestra el comienzo de una sección paralela. El código que esté dentro de esta sección será ejecutado por múltiples hilos de manera paralela.
- *`#pragma omp for`*: Esta directiva divide un bucle entre los hilos para su ejecución paralela.

A la hora de paralelizar un programa, también es fundamental gestionar correctamente el acceso a los datos. Para ello, OpenMP proporciona un conjunto de cláusulas que se utilizan junto con las directivas para indicar cómo deben tratarse las variables dentro de una región paralela. Las cláusulas se añaden a las directivas y

permiten especificar qué variables deben ser compartidas entre los hilos y cuáles deben ser privadas. Los ejemplos más comunes son:

- *shared*: Si se especifica que las variables son shared (compartidas), entonces todos los hilos pueden leer y escribir dichas variables. Este es el comportamiento por defecto.
- *private*: Cada hilo tiene su propia copia del valor de la variable, por lo que, aunque otro hilo la modifique, el resto de los hilos no ven este cambio en su copia local.

Además de las directivas, OpenMP permite el uso de variables de entorno, que permiten configurar aspectos clave de la ejecución del programa. Por ejemplo, con la variable de entorno `OMP_NUM_THREADS` podemos decirle al runtime de OpenMP que número de threads queremos de manera manual, si no se especifica el valor de estas variables de entorno, se usan los valores por defecto.

1.4. Vectorización

Tradicionalmente, los procesadores seguían un modelo de ejecución conocido como SISD (Single Instruction, Single Data), en el cual cada instrucción opera sobre un único dato a la vez. Para superar las limitaciones de SISD se emplea la vectorización, que se basa en realizar una sola instrucción sobre múltiples datos, a este tipo de arquitecturas específicas se les denomina SIMD (single instruction, multiple data), como se muestra en la figura 3.

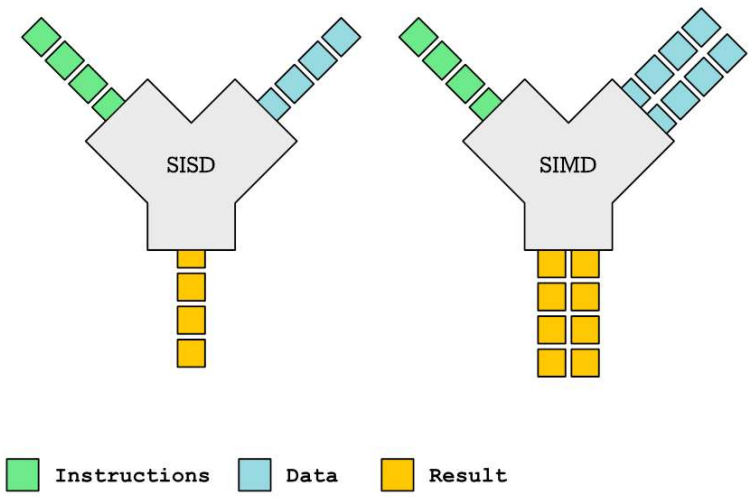

Figura 3: Comparación entre SISD y SIMD[4].

La vectorización es una técnica de programación que consiste en transformar bucles o secuencias de operaciones repetitivas para que puedan ejecutarse aprovechando el paralelismo de datos mediante la ejecución de instrucciones vectoriales o SIMD. Estas instrucciones permiten aplicar una misma operación (por ejemplo, suma, resta o multiplicación) simultáneamente sobre múltiples elementos de datos, en lugar de hacerlo de forma secuencial. De este modo, se aprovecha mejor el hardware disponible, ya que una sola instrucción puede operar sobre un vector de datos, lo que se traduce en una mayor paralelización a nivel de datos y una reducción en el número de ciclos de CPU necesarios para completar una tarea.

Las arquitecturas modernas cuentan con unidades vectoriales, capaces de realizar instrucciones SIMD. Algunos ejemplos de estas son:

- Intel con las extensiones vectoriales SSE, AVX, AVX2 y AVX-512. Estas instrucciones permiten realizar operaciones con vectores desde 128 bits hasta 512 por instrucción.
- AMD utiliza sus propias variantes compatibles con AVX.
- ARM emplea NEON en arquitecturas de 32 bits y SVE (Scalable Vector Extension) en sus procesadores de 64 bits, pensados para HPC.

Las instrucciones SIMD a diferencia de las instrucciones escalares pueden emplear registros vectoriales, que son registros especiales del procesador que, en vez de almacenar un solo elemento, almacenan varios elementos dependiendo de la arquitectura. Esto difiere del paradigma de los registros escalares tradicionales que solo almacenan un elemento por registro.

Para que dichas instrucciones resulten eficientes, es fundamental que los datos estén bien organizados en memoria. Estas instrucciones operan de forma simultánea sobre múltiples elementos, lo que incrementa la presión sobre la jerarquía de memoria. Dado que se procesan más datos por instrucción, se incrementa la tasa de accesos a memoria y, por tanto, la probabilidad de fallos de caché por instrucción de cómputo también aumenta. Esto puede reducir la capacidad del sistema para ocultar latencias mediante técnicas de prefetching o paralelismo de instrucciones.

Además, aunque estas suelen explotar de forma implícita la localidad espacial (al acceder a bloques contiguos de memoria), el rendimiento puede verse comprometido si los datos no están bien alineados o si el patrón de acceso no es contiguo, por ejemplo, en accesos dispersos o indirectos, donde no se aprovecha todo el ancho del vector. En estos casos, se incurre en penalizaciones por accesos no coalescentes, menor aprovechamiento del ancho de banda efectivo y mayores latencias de carga.

1.5. Utilización de la extensión SIMD: OpenMP y autovectorización

El uso manual de instrucciones vectoriales (SIMD) es complejo porque requiere trabajar con funciones intrínsecas del compilador (intrinsics), que permiten acceder directamente a instrucciones específicas del hardware, como SSE o AVX. Esto conlleva que el código sea menos legible, más difícil de mantener y menos portable. Además, el programador debe encargarse de aspectos como la alineación de memoria, el manejo de dependencias entre datos y la gestión precisa del tamaño de los vectores, lo cual puede ser propenso a errores.

Vectorizar a mano es difícil, por lo que existen herramientas para facilitarlo, OpenMP cuenta con directivas SIMD, que permiten al programador añadir de manera no invasiva secciones que vectorizarán bucles u operaciones. Aparte de vectorización manual, también hay maneras de vectorizar utilizando flags de compilador. Lo que hacen dichas flags, es decirle al compilador que analice el código y busque regiones que puedan ser vectorizables y así, aprovechar las unidades vectoriales del procesador.

La principal directiva OpenMP para gestionar la vectorización es la siguiente:

- *`#pragma omp simd`*: Es la directiva más útil, ya que, permite indicar que bucle queremos que se vectorice. De esta forma, el compilador intentará generar una única instrucción vectorial que sume varios elementos al mismo tiempo aprovechando los registros vectoriales.

Después, al igual que con el paralelismo de hilos, tenemos cláusulas que modifican el comportamiento de las directivas, las más utilizadas son:

- *`simdlen(tamaño)`*: Especifica el número de iteraciones que se ejecutarán concurrentemente, es decir, especifica el número de elementos que hay en el vector que se está utilizando.
- *`collapse(n)`*: Esta cláusula se utiliza junto con la directiva `simd` de OpenMP y especifica cuántos bucles anidados deben tratarse como un único bucle plano para aplicar la vectorización. Es útil cuando existen varios bucles anidados que pueden recorrerse de forma independiente. Al indicarse, el compilador combinará los n primeros bucles anidados en un solo bucle con un rango más amplio, lo que permite aplicar instrucciones SIMD sobre un espacio mayor.

Para la autovectorización, tenemos la opción `–ftree-vectorize`, que está disponible en compiladores como GCC y Clang. Lo que hace esta flag es analizar bucles y funciones iterativas, transformándolas en instrucciones SIMD si es posible hacerlo. La idea es que, con esta flag sea automáticamente el compilador el que se encargue de

vectorizar y no el usuario. En compilaciones con niveles de optimización –O2 y –O3 suele estar activada, pero se puede forzar manualmente con `–ftree-vectorize`.

Otra flag a mencionar es `–march=`, que permite seleccionar la arquitectura concreta del procesador en el que estamos compilando. De esta forma, el compilador es capaz de optimizar al máximo las instrucciones vectoriales, para que, se adapten al hardware destino. Por ejemplo, la flag `–march=sapphirerapids` le dice al compilador que la arquitectura en la que se está compilando es de la familia Sapphire Rapids, una arquitectura de la gama Xeon de cuarta generación. Dicha arquitectura cuenta con AVX-512 y AMX (Advanced Matrix Extensions), lo que permite trabajar con bloques aún mayores y de manera más eficiente.

2. Metodología

En este capítulo se describe la metodología seguida para el desarrollo del proyecto. En primer lugar, se define el propósito de la experimentación y los criterios empleados, seguido de un apartado con todos los benchmarks utilizados. A continuación, se detalla la configuración, el entorno en el que se ha realizado el proyecto y parte del software empleado, incluyendo las herramientas de monitorización, el compilador y sus flags. Posteriormente, se presenta el volumen de trabajo gestionado. Y por último, se describen las principales métricas extraídas mediante VTune, que se han utilizado para el análisis del rendimiento y comportamiento de cada aplicación.

2.1. Objetivo de la experimentación

El propósito principal del proyecto es comparar como diferentes implementaciones de un mismo problema, dependiendo de diferentes factores a la hora de paralelizar, pueden afectar a su rendimiento. Para ello, se han seleccionado 9 benchmarks con diferentes patrones computacionales, lo que permite observar cómo los diferentes tipos de paralelismo afectan al rendimiento dependiendo de la carga de trabajo. El análisis se centra especialmente en tres dimensiones clave de optimización:

- *Paralelismo a nivel de hilos*: Utilizando la API de OpenMP.
- *Vectorización*: Empleando directivas SIMD de OpenMP para evaluar diferentes longitudes de vector.
- *Autovectorización mediante opciones del compilador*: Tanto mediante opciones de compilación de GCC o G++, como incluyendo configuraciones adaptadas al hardware (por ejemplo, con instrucciones AVX).

Cada benchmark se ha modificado en varias versiones que difieren entre ellas en aspectos clave, como el número de hilos utilizados, la presencia o no de vectorización, y la estrategia empleada para aplicarla (ya sea mediante compilación automática, directivas o vectorización haciendo uso de OpenMP). La carga de trabajo y los parámetros de entrada se han mantenido constantes en todas las implementaciones de cada benchmark. Esto permite garantizar una base común para comparar y observar con mayor precisión cómo influyen las distintas técnicas de paralelización y vectorización en el rendimiento de las aplicaciones. El objetivo es analizar si existe una relación directa entre estas decisiones de implementación y el comportamiento observado.

De este modo, lo que se busca con esta experimentación es analizar benchmarks que tienen patrones fundamentales de computación y aplicaciones diversas, para de esta forma, evaluar qué factores son los que más repercusión tienen en el rendimiento computacional.

2.2. BENCHMARKS

Para el análisis de rendimiento de este proyecto, se han seleccionado distintas aplicaciones que operan en dominios diversos. Con el fin de obtener una visión más completa, los benchmarks utilizados están diseñados para ejercer presión sobre distintas partes del sistema computacional, lo que permite identificar en qué casos resulta relevante aplicar técnicas de vectorización y en cuáles no.

Para cubrir un amplio espectro de patrones computacionales y cargas de trabajo, se han elegido un total de nueve benchmarks procedentes de distintas suites de evaluación como se muestra en la tabla 2.

Rodinia [5]	Backprop (Backpropagation)
	BFS (Breadth-First Search)
	CFD (Computational Fluid Dynamics)
	Hotspot
	LavaMD
Polybench-ACC [6]	GEMM (General Matrix-Matrix Multiplication)
	Jacobi 2d
	Convolution 3d
SpMV (Sparse Matrix-Vector Multiply) [7]	

Tabla 2: Suites y sus respectivos benchmarks

Los Berkeley Dwarfs constituyen una taxonomía desarrollada por la Universidad de California, Berkeley, que agrupa los principales patrones computacionales presentes en aplicaciones científicas, de simulación y procesamiento de datos. Cada dwarf representa una clase de cómputo con características comunes en cuanto a paralelismo y patrones de acceso a memoria. Esta clasificación es especialmente útil para interpretar el rendimiento de los benchmarks, ya que permite establecer expectativas sobre su escalabilidad, eficiencia en el uso de recursos y respuesta ante técnicas de optimización. En el contexto de este trabajo, los dwarfs adquieren una relevancia particular, dado que cada benchmark analizado representa un patrón computacional distinto, lo que proporciona una base variada y representativa para realizar un análisis comparativo sólido y generalizable. La explicación detallada de cada benchmark y su correspondencia con los patrones de los Berkeley Dwarfs se presenta en los apartados siguientes.

2.2.1. Backprop

El algoritmo que utiliza consta de dos fases principales. En primer lugar, la propagación hacia adelante: los datos de entrada se introducen en la red neuronal, donde se procesan a través de una o más capas mediante operaciones matriciales, obteniendo como resultado una salida (predicción). Esta salida se compara con el valor esperado, y a partir de esa comparación se calcula un error. En la segunda parte, la fase de retropropagación, lo que hace es propagar el error desde la salida a capas anteriores, de esta manera, determinando cuánto ha contribuido cada peso al error total. Con esta información, se actualizan los pesos de la red.

Desde un punto de vista de patrones computacionales y siguiendo la taxonomía Berkeley Dwarfs, Backprop se clasifica principalmente dentro del algebra lineal densa (Dense Linear Algebra). Esto se debe a que Backprop se basa principalmente de multiplicaciones y sumas entre vectores y matrices densas. Además, también se podría discutir que al usar accesos regulares a memoria y con patrones predecibles, se podría agrupar dentro de cuadricula estructurada (Structured Grid), pero esto sería de manera secundaria.

En cuanto a su dominio de aplicación, Backprop pertenece al campo de la inteligencia artificial y el aprendizaje automático (machine learning), en concreto al entrenamiento supervisado de redes neuronales.

2.2.2. BFS

El objetivo del algoritmo es recorrer un grafo partiendo de un nodo llamado origen y explorar todos los nodos alcanzables en un orden basado en la distancia mínima al origen. El algoritmo opera por niveles: primero, se procesan los vecinos directos del nodo inicial, es decir, los nodos que se encuentran a una distancia de un salto, en la siguiente iteración se visitan los nodos vecinos de esos otros nodos, los que se encuentran a dos saltos del nodo origen y así sucesivamente hasta recorrer todo el grafo.

Desde el punto de vista de la taxonomía de patrones computacionales Berkeley Dwarfs, BFS se clasifica principalmente dentro del patrón Graph Traversal, el cual se caracteriza por estructuras de datos irregulares, accesos a memoria no contiguos y una alta dependencia de los datos para determinar el flujo de la ejecución. No obstante, algunas implementaciones pueden presentar ciertos aspectos del patrón Unstructured Grid, sobre todo cuando los accesos a memoria presentan alguna regularidad o si el grafo tiene una estructura algo homogénea.

2.2.3. CFD

El algoritmo principal de CFD (Computational Fluid Dynamics) consiste en iterar sobre una malla formada por una matriz y aplicar un conjunto de operaciones en cada celda para calcular el flujo neto a través de sus caras. El sistema simula cómo las propiedades del fluido cambian dentro de cada volumen a medida que el flujo atraviesa dicha malla. Estas operaciones incluyen cómputos intensivos en punto flotante como interpolaciones, diferencias de valores, productos vectoriales, normalización y evaluación de funciones físicas.

Desde el punto de vista de los patrones computacionales definidos por la taxonomía Berkeley Dwarfs, CFD se clasifica dentro del patrón Unstructured Grid, dado que opera sobre una malla no estructurada en la que las conexiones entre celdas no siguen un patrón uniforme. Este tipo de patrón se caracteriza por un acceso irregular a memoria, una dependencia fuerte de la conectividad del dominio y conlleva un problema para la vectorización o paralelismo masivo de forma eficiente.

En cuanto al dominio de aplicación, CFD pertenece al campo de la simulación científica y la ingeniería computacional. Las simulaciones de dinámica de fluidos se utilizan ampliamente en contextos donde es necesario modelar el comportamiento de líquidos y gases.

2.2.4. Convolution 3D

El kernel realiza una operación de convolución sobre un matriz tridimensional, es decir, recorre un volumen de datos y calcula un nuevo valor para cada punto de la matriz tridimensional de salida. Para hacerlo, toma un vecindario en forma de cubo de tamaño 3x3x3 alrededor de cada punto de la matriz tridimensional de entrada (excepto en los bordes), y combina esos 27 valores mediante una suma ponderada con una máscara predefinida. Esta operación se repite para todos los puntos de la matriz.

Desde la perspectiva de la taxonomía Berkeley Dwarfs, Convolution 3D se clasifica dentro del patrón Structured Grid, ya que opera sobre un dominio regular y multidimensional, con cálculos repetitivos y vecinos bien definidos. También puede considerarse como una instancia del patrón Stencil, muy común en simulaciones numéricas.

Este benchmark es representativo de algoritmos reales que aparecen en deep learning, procesamiento de imágenes volumétricas, y modelado físico donde se aplican operadores espaciales sobre datos tridimensionales.

2.2.5. GEMM

El algoritmo GEMM (General Matrix-Matrix Multiplication) realiza una operación típica de multiplicación de matrices, en la que se combinan tres matrices densas: A, B y C. El cálculo consiste en multiplicar A por B y por un valor alpha, y sumarlo a la matriz C, también multiplicada por un valor beta. Esta operación se lleva a cabo recorriendo las matrices con bucles anidados.

En cuanto a los patrones computacionales según la taxonomía de los Berkeley Dwarfs, GEMM pertenece al patrón Dense Linear Algebra, caracterizado por operaciones aritméticas intensivas sobre estructuras matriciales densas. Este patrón se distingue por un acceso regular a memoria, buen aprovechamiento de la jerarquía de caché y un potencial elevado para vectorización y paralelismo.

Desde el punto de vista de la aplicación, la multiplicación de matrices aparece en un sinfín de contextos, por ejemplo: simulaciones físicas, procesamiento de imágenes, visión por computador y aprendizaje automático, entre otros.

2.2.6. Hotspot

El algoritmo de Hotspot se basa en resolver una ecuación de conducción de calor sobre una malla. Cada celda intercambia calor con sus celdas vecinas en las direcciones norte, sur, este y oeste, lo cual genera una carga computacional relativamente sencilla pero intensiva, donde hay operaciones aritméticas sobre matrices y accesos regulares a memoria.

Desde el punto de vista de los patrones computacionales de la taxonomía Berkeley Dwarfs, HotSpot se clasifica dentro del patrón Structured Grid, ya que opera sobre una malla regular con conectividad fija entre elementos vecinos. Este tipo de patrón se caracteriza por accesos a memoria predecibles, alta localidad espacial y facilidad para vectorizar.

En cuanto a su dominio de aplicación, Hotspot se encuentra en la intersección entre la arquitectura de computadores y la simulación térmica.

2.2.7. Jacobi 2D

El algoritmo de Jacobi 2D realiza una iteración sobre una matriz de dos dimensiones de valores escalares, donde cada punto del dominio se actualiza utilizando el promedio de sus vecinos directos (norte, sur, este y oeste). Esta operación se repite múltiples veces, representando una evolución artificial del sistema hacia una solución estable, es decir, que ya no cambien mucho entre una iteración y otra.

Desde la perspectiva de la taxonomía de Berkeley Dwarfs, Jacobi 2D pertenece al patrón Stencil. Este tipo de patrones implica accesos regulares a memoria, lo que significa que el rendimiento está frecuentemente limitado por el ancho de banda de memoria y no por la capacidad de cálculo.

En el contexto de aplicaciones reales, métodos tipo Jacobi se utilizan como, por ejemplo, en aplicaciones en simulaciones térmicas, difusión y dinámica de fluidos.

2.2.8. LavaMD

El algoritmo de lavaMD calcula interacciones entre un número determinado de átomos distribuidos en una malla tridimensional. Para cada átomo, se calcula la fuerza resultante debida a la interacción con sus vecinos dentro de una región. Las fuerzas se modelan en esta versión concreta con un modelo sencillo enfocado a resaltar patrones computacionales relevantes, es decir, es una versión simplificada del algoritmo.

LavaMD realiza tres operaciones fundamentales:

- Calcula distancias entre pares de partículas.
- Evalúa fuerzas basadas en esas distancias.
- Acumula los resultados para actualizar propiedades como la aceleración o la energía potencial.

Desde la perspectiva de la taxonomía Berkeley Dwarfs, lavaMD se clasifica dentro del patrón Structured Grid, debido a que opera sobre un dominio regular, con vecinos bien definidos y patrones repetitivos y que no varían localmente. Además, también incorpora elementos del patrón N-Body, ya que cada partícula interactúa con otras, pero en este caso está restringido a vecinos locales.

LavaMD es representativo de aplicaciones reales en el campo de la simulación de sistemas físicos, especialmente en dinámica molecular, modelado de materiales, biología computacional y química teórica.

2.2.9. SpMV

El algoritmo SpMV en formato CSR multiplica una matriz dispersa por un vector denso. Para cada fila, se recorren únicamente sus elementos no nulos y se calcula la suma del producto entre cada valor y su correspondiente entrada del vector. El resultado se almacena en la posición correspondiente del vector de salida.

Desde el punto de vista de la taxonomía de Berkeley Dwarfs, SpMV se clasifica dentro del patrón Sparse Linear Algebra, caracterizado por operaciones sobre estructuras dispersas con un alto grado de irregularidad en memoria. Este patrón

impone desafíos particulares por la baja localidad espacial y temporal de los datos, que tiende a penalizar el rendimiento.

En el contexto de aplicaciones reales, SpMV puede aparecer en distintos contextos, por ejemplo: simulaciones físicas, procesamiento de grafos, machine learning y análisis estructural.

2.3. Configuración experimental

Todos los experimentos realizados en este proyecto se han llevado a cabo en el supercomputador Tritón, una infraestructura de altas prestaciones perteneciente al grupo de Arquitectura y Tecnología de Computadores de la Universidad de Cantabria. El uso de Tritón permite unas condiciones estables de ejecución y potencia computacional suficiente, lo que permite obtener resultados reproducibles y representativos.

Durante las pruebas, se ha empleado el sistema SLURM (Simple Linux Utility for Resource Management), un planificador de trabajos muy utilizado en entornos de cómputo de alto rendimiento, que gestiona la asignación de recursos [8]. Mediante opciones como `--exclusive`, se garantiza el uso del nodo en exclusiva, sin que otros trabajos perturben las mediciones.

Para ello, se ha seleccionado el nodo comp81, perteneciente al supercomputador Tritón. Este nodo está equipado con dos procesadores Intel Xeon Gold 6416H [9], cada uno con 18 núcleos físicos y 36 hilos gracias a la tecnología Hyper-Threading. La frecuencia base es de 2,20 GHz, y puede alcanzar hasta 4,20 GHz en modo Turbo. Su jerarquía de memoria cuenta con 32 KiB de caché L1 por núcleo, 2 MiB de L2 por núcleo, y 45 MiB compartidos de caché L3. Además, dispone de 384 GB de memoria RAM, lo cual asegura una plataforma potente y adecuada para evaluar cargas de trabajo paralelas y optimizaciones vectorizadas.

Cada benchmark se ha ejecutado de manera independiente y aislada del resto, pero todos comparten aspectos comunes relativos a la configuración de sus ejecuciones, estos aspectos son:

- *Número de hilos*: Para tener una visión general del comportamiento se han utilizado 1, 4, 8, 16 y 32 hilos para todas las ejecuciones, excepto para el benchmark de SpMV, que ha requerido la adición de un punto medio entre 1 y 4 hilos, que ha sido 2. Los valores intermedios han sido elegidos con una separación progresiva que permite analizar el comportamiento de escalabilidad de forma clara y representativa.
- *Versiones*: Todos los benchmarks cuentan con mínimo 3 ejecutables, estos son: la versión base sin vectorizar, la versión vectorizada con las herramientas

del compilador de GCC o G++, etiquetadas en los resultados con el sufijo GCC y la versión vectorizada por el compilador sabiendo la estructura del computador, con el sufijo AVX. Hay casos en los que se ha podido vectorizar manualmente porque el código lo permitía. En dichos ejecutables se ve marcado el tamaño de vector a continuación del nombre, siendo el número de elementos del vector 1, 4 y 8.

- *Parámetros de entrada*: cada uno de los benchmarks tiene sus propios parámetros de entrada, que se han adaptado para garantizar un tiempo de ejecución representativo en el que el análisis de rendimiento tenga sentido. El único parámetro de entrada común para todos los benchmarks es el número de hilos que queremos que se ejecuten.

Para garantizar una coherencia en las mediciones de rendimiento de todos los benchmarks, se ha empleado el mismo tipo de análisis en Vtune para todos. En concreto, se ha empleado el perfil Memory Access, que permite evaluar el comportamiento de la jerarquía de memoria, identificar cuellos de botella relacionados con el acceso a datos, y analizar métricas como las tasas de fallo en caché (L1, L2 y LLC), el uso de la CPU y el CPI. Aparte de centrarse en el comportamiento de la memoria, permite el uso de contadores hardware muy precisos y útiles para el posterior análisis de los resultados.

2.4. Automatización de la ejecución

El principal objetivo de la automatización en este proyecto ha sido facilitar la ejecución consecutiva de todas los ejecutables de un mismo benchmark de forma controlada y sin intervención manual. Esto permite reducir significativamente la posibilidad de cometer errores al introducir parámetros, ya que todos se gestionan de forma sistemática y reproducible. Además, al ejecutarse todo de manera secuencial y organizada, se garantiza una mayor consistencia en las condiciones de prueba, minimizando las posibles variaciones accidentales entre ejecuciones. Por último, esta automatización permite ahorrar una cantidad considerable de tiempo al eliminar la necesidad de repetir manualmente cada combinación de prueba, lo que resulta especialmente beneficioso cuando se trabaja con múltiples benchmarks y configuraciones. En la figura 4, se muestra un ejemplo de todos los directorios generados para observar la magnitud del proyecto.

Los scripts utilizados han sido los siguientes siete y están disponibles en un repositorio de git[10]:

- *tfg.sh*: Este script actúa como job script para SLURM, es decir, un archivo que define los comandos necesarios para enviar un trabajo al planificador de recursos. Por lo tanto, se encarga de gestionar la solicitud y asignación de recursos en el nodo de cómputo. Especifica los parámetros necesarios para la ejecución del trabajo y, una vez asignados los recursos, lanza el script principal (principal.py) mediante el comando `srun`.
- *principal.py*: Este script es el encargado de gestionar la ejecución automatizada de los experimentos. Recoge como argumentos de entrada el nombre del benchmark y el parámetro `--outputdir`, que indica el nombre del directorio principal donde se almacenará toda la información relacionada con los análisis. Este nombre sigue un formato tipo fecha-benchmark, lo que facilita la identificación y organización de los resultados. Además, se encarga de coordinar al resto de scripts y pasarles sus argumentos de entrada.
- *ejecutarBenchmarks.py*: Este script recibe como parámetros un array de ejecutables, que son las distintas versiones de un solo benchmark y el directorio principal donde ha de dejar los resultados. Este script ejecuta los benchmarks recibidos utilizando VTune desde la línea de comandos. De esta forma, se facilita la automatización de la recogida de resultados. Todos estos ficheros generados van a distintas carpetas donde se almacenan dependiendo del ejecutable al que pertenecen. Aquí se guardarán en subdirectorios dependiendo del número de hilos.
- *transformarCSV.py*: Este script coge todos los resultados de las ejecuciones y se encarga de generar un archivo del tipo .csv mediante un comando de VTune, al que se le pasa en este script la opción `summary`, que muestra

métricas clave como el tiempo de ejecución y los cuellos de botella más relevantes. Esto permite extraer información de manera más sencilla a la hora de hacerlo de forma automática. Todos los CSV van a una carpeta del tipo CSV_benchmark.

- *transformarCSVHW.py*: Al igual que transformarCSV.py, este script se encarga de transformar los análisis en `.csv`. En este caso, estos ficheros muestran contadores hardware más complejos, al utilizar la opción `hardware-events`, que permite obtener información a bajo nivel a partir de contadores hardware. Los .csv en este caso se almacenan en un directorio con la estructura CSV_benchmarkHW.
- *crearGraficas.py*: Este script es el encargado de generar todas las gráficas para su posterior análisis. Para generar dichas graficas, coge los archivos .csv generados por transformarCSV.py y extrae la información requerida, que después será depositada en un directorio del tipo graficas_benchmark.
- *crearGraficasHW.py*: Al igual que crearGraficas.py, este script genera graficas a partir de .csv, pero en este caso, las gráficas son referentes a los .csv de transformarCSVHW.py. Los resultados van a un directorio diferente a los de crearGraficas.py, en este caso se llama graficasHW_benchmark.

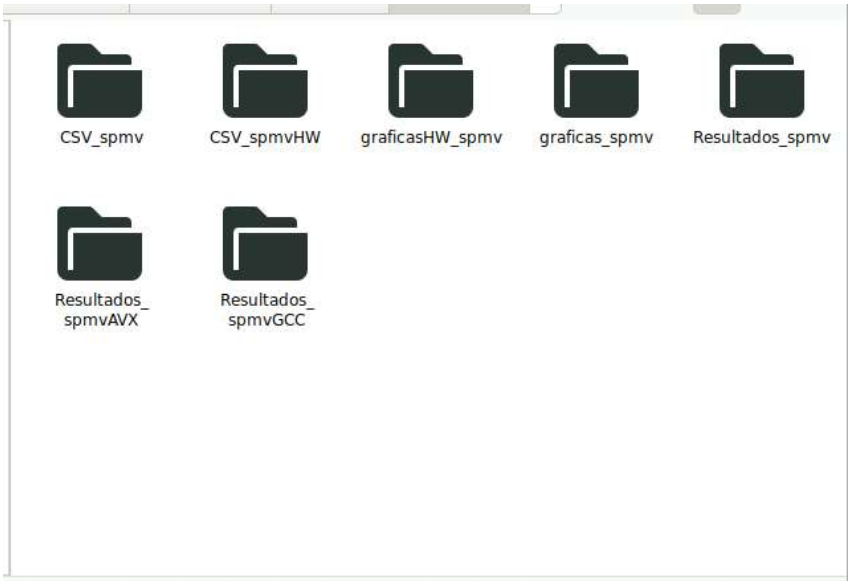

Figura 4: Estructura principal de los directorios.

2.5. Escenarios evaluados

A lo largo de este proyecto se ha trabajado con un total de nueve benchmarks, seleccionados por representar distintos patrones computacionales y dominios de aplicación, tal y como se describe en el capítulo 1. El objetivo ha sido cubrir una variedad de comportamientos en cuanto a paralelismo, acceso a memoria y potencial de vectorización, con el fin de evaluar cómo diferentes técnicas de optimización afectan al rendimiento.

Cada benchmark ha sido ejecutado en múltiples variantes, lo que ha generado un alto volumen de ejecuciones, como se puede comprobar en la tabla 3.

Benchmark	Ejecutables
Backprop	• backprop • backpropGCC • backpropAVX
BFS	• bfs • bfsGCC • bfsAVX
CFD	• euler3d_cpu1 • euler3d_cpu4 • euler3d_cpu8 • euler3d_cpu GCC • euler3d_cpu AVX
Convolution3D	• convolution-3d • convolution-3dGCC • convolution-3dAVX
GEMM	• gemm_acc • gemm_accGCC • gemm_accAVX
Hotspot	• hotspot1 • hotspot4 • hotspot8 • hotspotGCC • hotspotAVX
Jacobi2D	• jacobi-2d-imper_acc • jacobi-2d-imperGCC • jacobi-2d-imperAVX
LavaMD	• lavamd • lavamdGCC • lavamdAVX
SpMV	• spmv • spmvGCC • spmvAVX

Tabla 3: Ejecutables junto a sus respectivos benchmarks.

Cada una de estas aplicaciones se ha ejecutado con distintas configuraciones de hilos mencionadas en el apartado 2.3. Esto implica que, por cada benchmark, se han realizado al menos 15 ejecuciones (3 ejecutables × 5 configuraciones), y en algunos casos, hasta 25 ejecuciones cuando hay más variantes vectorizadas.

Además, cada benchmark se ha ejecutado entre seis y siete veces, seleccionando manualmente el resultado más representativo para su análisis. En algunos casos, ha sido necesario repetir ejecuciones debido a errores puntuales, variaciones inesperadas o problemas en la recopilación de datos, con el objetivo de asegurar que los resultados fueran consistentes y válidos para su posterior análisis.

2.6. Métricas

Con el objetivo de evaluar el impacto de las distintas técnicas de paralelización y vectorización aplicadas a los benchmarks, se han recogido y analizado diversas métricas de rendimiento utilizando VTune. Para ello, se ha empleado el análisis tipo Memory Access y como se menciona en el apartado 3 de este mismo capítulo, los datos de los perfiles `summary` y `hardware-events`, que permiten obtener métricas tanto generales como específicas a nivel de microarquitectura.

Este análisis proporciona un resumen general del comportamiento de la aplicación, permitiendo observar tanto el rendimiento global como el impacto de la jerarquía de memoria. Los datos se han exportado a formato .csv y procesado automáticamente mediante los scripts desarrollados en el proyecto.

Las métricas más relevantes extraídas en esta categoría son:

Métrica	Descripción
Elapsed Time	Tiempo total de ejecución (en segundos).
DRAM Bound	Porcentaje del tiempo en que la CPU está esperando datos desde la memoria principal (DRAM).
L1 Bound	Porcentaje del tiempo en que la ejecución está limitada por el acceso a caché de primer nivel.
L2 Bound	Porcentaje de tiempo en que el rendimiento se ve afectado por la caché L2.
L3 Bound	Porcentaje del tiempo afectado por esperas en la caché de último nivel (LLC).

Tabla 4: Métricas utilizadas en el perfil summary.

Además del perfil `summary`, se ha realizado un análisis adicional a través de eventos hardware específicos. Estas métricas no se encuentran directamente en los

.csv, sino que han sido calculadas manualmente a partir de los contadores exportados por VTune mediante el modo `hardware-events`. Este análisis proporciona una visión más detallada del comportamiento interno del procesador durante la ejecución de cada benchmark.

Las métricas clave derivadas de los contadores hardware son:

Métrica	Descripción
CPI (Cycles/Instr.)	Número medio de ciclos necesarios por instrucción ejecutada.
L1Miss Rate	Tasa de fallos de caché L1 respecto al total de accesos.
L2 Miss Rate	Tasa de fallos de caché L2.
L3 Miss Rate	Tasa de fallos en la caché de último nivel.

Tabla 5: Métricas utilizadas en el perfil hardware-events.

Estas métricas han sido seleccionadas no solo para comprender el tiempo de ejecución de los benchmarks, sino también qué elementos del sistema limitan el rendimiento, como la memoria principal, la jerarquía de caché o la eficiencia de uso del procesador. Al analizar estos valores como un conjunto, se ha podido realizar un estudio más completo sobre el impacto de los distintos tipos de paralelización.

A continuación, se explica la motivación detrás de la selección de cada una de las métricas:

- *Elapsed Time*: se ha seleccionado como métrica principal de rendimiento, ya que refleja directamente el tiempo que tarda una aplicación en completarse. Es la medida más inmediata y refleja el impacto real de las optimizaciones tanto al aumentar el número de hilos como las posibles mejoras que muestre la vectorización.
- *DRAM Bound, L1 Bound, L2 Bound y L3 Bound*: se han incluido para identificar cuellos de botella en la jerarquía de memoria, ya que, miden el tiempo que la jerarquía de memoria impacta la ejecución. Estas métricas ayudan a entender si el rendimiento está limitado por el acceso a los distintos niveles de caché o a la memoria principal, lo cual es especialmente relevante en aplicaciones que trabajan con grandes volúmenes de datos o tienen patrones de acceso poco eficientes.
- *CPI (ciclos por instrucción)*: permite evaluar la eficiencia con la que se ejecutan las instrucciones y detectar posibles cuellos de botella en la jerarquía de memoria. Un CPI alto suele indicar que la CPU pasa muchos ciclos sin ejecutar instrucciones, lo que puede deberse a problemas de memoria, dependencia de datos o mal aprovechamiento de la vectorización.

- *L1, L2 y L3 Miss Rate*: proporcionan información detallada sobre la frecuencia con la que, al solicitar datos a la jerarquía de memoria, no se encuentran en cada uno de los niveles de caché. Estas métricas son útiles para correlacionar el comportamiento observado con los valores de bound, y permiten confirmar si los fallos de caché están afectando significativamente al rendimiento.

Además de las métricas presentadas, se había considerado inicialmente la energía consumida como un factor adicional para evaluar el rendimiento de las distintas implementaciones. Sin embargo, por limitaciones técnicas al utilizar Intel VTune, no fue posible recopilar datos de consumo energético de forma fiable en el entorno experimental utilizado. Aun así, la energía es una métrica muy relevante, especialmente en arquitecturas modernas donde el rendimiento por vatio es crítico, por lo que su inclusión sería una mejora interesante para futuros trabajos o extensiones de este proyecto.

Por motivos de espacio, en la sección de análisis solo se mostrarán las métricas más relevantes en cada caso, priorizando aquellas que ofrecen mayor valor comparativo entre las distintas variantes analizadas.

3. Análisis

En este capítulo se presenta el análisis de los resultados obtenidos tras la ejecución de los diferentes benchmarks descritos en los capítulos anteriores. Cada uno de ellos ha sido evaluado en distintas versiones, variando el número de hilos al paralelizar, el tipo de vectorización y los flags del compilador utilizado. Esto con el fin de estudiar cómo estas decisiones de implementación influyen en el rendimiento computacional.

El análisis se realiza de manera individual por benchmark, permitiendo observar con detalle cómo responde cada aplicación a las distintas configuraciones experimentales. Para ello, se emplean tanto métricas como el tiempo de ejecución, de acceso a memoria y eficiencia a nivel de microarquitectura, obtenidas a través de Intel VTune Profiler.

Dado que los benchmarks utilizados representan distintos patrones computacionales, es importante destacar su clasificación dentro de la taxonomía Berkeley Dwarfs, lo cual permite interpretar mejor su comportamiento y compararlos en función de sus características fundamentales. La siguiente tabla resume esta clasificación:

Benchmark	**Patrón computacional (Berkeley Dwarf)**
Backprop	Dense Linear Algebra *(secundariamente: Structured Grid)*
BFS	Graph Traversal *(potencialmente: Unstructured Grid)*
CFD	Unstructured Grid
Convolution 3D	Structured Grid / Stencil
GEMM	Dense Linear Algebra
Hotspot	Structured Grid
Jacobi 2D	Stencil
LavaMD	Structured Grid *(con elementos de N-Body)*
SpMV	Sparse Linear Algebra

Tabla 6: Benchmarks y sus correspondientes Berkeley Dwarfs[11].

Esta clasificación proporciona un marco de referencia útil para entender el comportamiento observado en los análisis posteriores, especialmente en lo que respecta a escalabilidad, eficiencia del uso de memoria y potencial de vectorización.

3.1. Backprop

Como se definió en el capítulo 2 apartado 2, Backprop es un algoritmo empleado para el entrenamiento de redes neuronales, en el que se calcula el error y se ajustan los pesos de la red mediante propagación hacia atrás. Involucra principalmente operaciones matriciales y cálculos repetitivos sobre grandes volúmenes de datos.

En cuanto a Backprop a nivel de jerarquía de memoria, CPI y tiempo de ejecución se observa:

- La gráfica de la figura 5, representa el porcentaje del tiempo de ejecución que se pierde en accesos a L1. En cuanto se empieza a hacer uso de los hilos, el uso de tiempo de CPU se incrementa de manera significativa para backprop y para backpropGCC. Por otra parte, backpropAVX sube de manera menos abrupta. Observando la gráfica de la tasa de fallos en L1 no se obtienen conclusiones claras de ella, ya que, la tasa de fallo máximo es inferior al 8%, lo que implica muchos accesos a L1, pero pocos fallos.
- El siguiente nivel de la jerarquía de memoria, el nivel 2, no muestra un porcentaje de consumo de tiempo significativo en ninguna de las implementaciones, con un máximo del 1,2%. La tasa de fallos, por otra parte, llega al 70%, pero como el consumo de tiempo en accesos a la L2 es tan pequeño, no es necesario hacer mucho énfasis en esta métrica.
- Por otra parte, en la figura 6 se observa que el nivel 3 de la jerarquía de memoria muestra un nivel de porcentaje de consumo de tiempo ascendente, siendo backprop y backpropGCC superiores a backpropAVX, excepto para 8 y 16 hilos, aunque la diferencia en estos casos es muy baja. En la figura 8, que muestra la tasa de fallos de L3, se observa una disminución que tiende a estabilizarse a partir de los 16 hilos, excepto para el caso de backprop. Esto implica que se reducen los fallos en la L3, pero aumentan los accesos a esta.
- La figura 7 muestra el tiempo que los accesos a memoria principal bloquean la ejecución. Se observa un comportamiento descendente, con una tendencia a estabilizarse cerca del 20%, excepto para backprop que muestra un pico en 24 hilos.
- La figura 9 muestra el tiempo de ejecución de las distintas versiones de Backprop y la figura 10 muestra el CPI (es decir, los ciclos de reloj por instrucción ejecutada). Ambas figuras muestran un comportamiento decreciente que tiende al estancamiento a partir de los 16 hilos, con un pico para backprop y backpropGCC en 4 hilos.

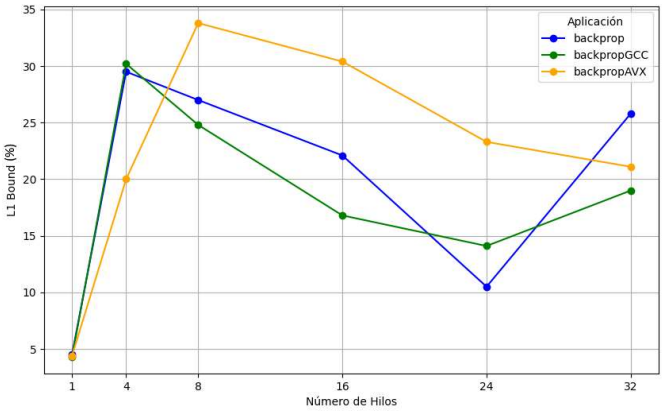

Figura 5: Gráfica que muestra una comparativa del porcentaje de tiempo que bloquea L1 la ejecución de Backprop. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

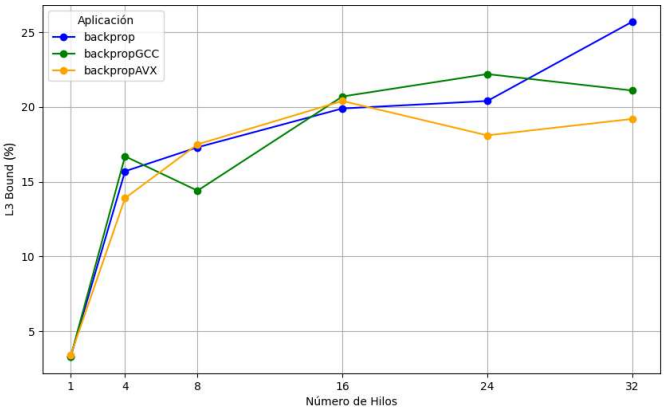

Figura 6: Gráfica que muestra una comparativa del porcentaje de tiempo que bloquea L3 la ejecución de Backprop. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

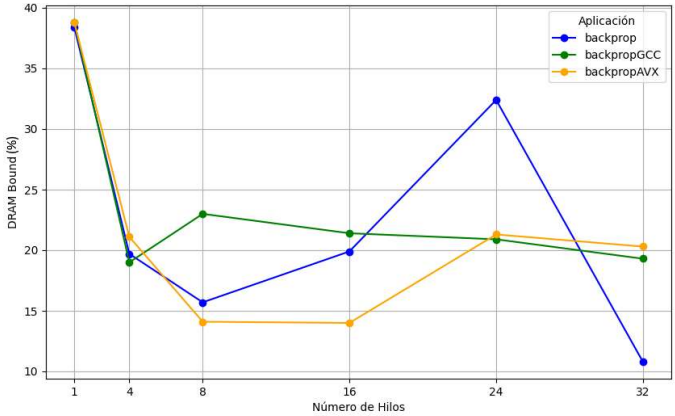

Figura 7: Gráfica que muestra una comparativa de la tasa de fallos en L3 de Backprop. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

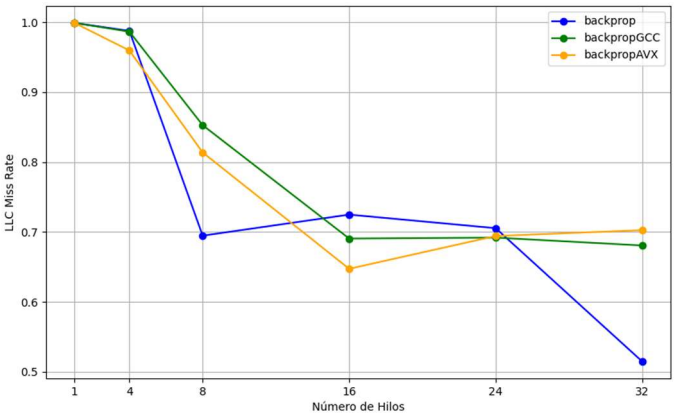

Figura 8: Gráfica que muestra una comparativa de la tasa de fallos en L3 de Backprop. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

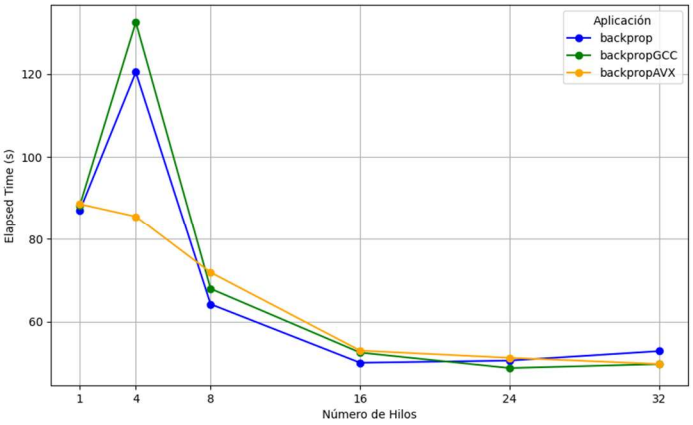

Figura 9: Gráfica que muestra una comparativa del tiempo de ejecución de Backprop. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

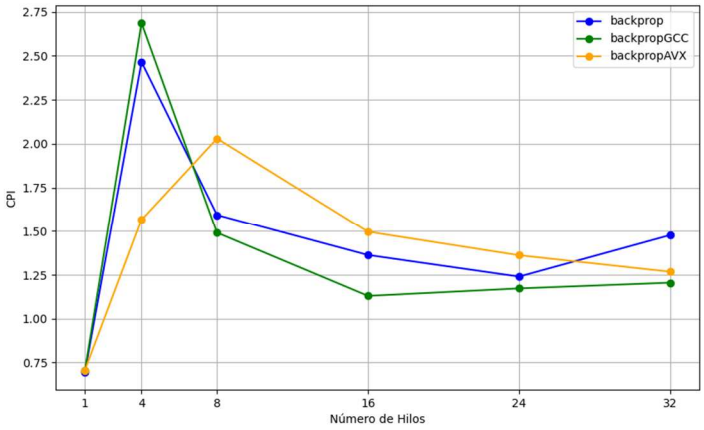

Figura 10: Gráfica que muestra una comparativa de los ciclos por instrucción que se ejecutan en Backprop. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

Como se puede observar en los resultados, Backprop es un benchmark que ejecuta muchísimas instrucciones al ser intensivo en cómputo, pero que también hace mucho uso de la jerarquía de memoria, sobre todo de los niveles más altos. Esto implica un CPI alto y un estancamiento en los tiempos de ejecución en 30 segundos a partir de 16 hilos. Como se puede observar también, hay un pico en tiempos de ejecución para 4 hilos, esto se debe a que las versiones backprop y backpropGCC acceden mucho a L1 en ese caso aislado. Aunque los algoritmos de álgebra lineal densa como Backprop suelen escalar bien con el número de hilos, en este caso la saturación de la jerarquía de memoria limita ese crecimiento. Esto es coherente con lo observado en las figuras.

En resumen, Backprop es un benchmark con alta intensidad computacional, pero cuya eficiencia está fuertemente condicionada por los accesos a memoria, especialmente en los niveles L1 y L3. La vectorización, aunque mejora ligeramente el comportamiento en caché, no reduce de forma significativa el CPI ni el tiempo de ejecución respecto a la versión no vectorizada. Esto sugiere que, en este caso, los cuellos de botella en memoria limitan el beneficio que puede obtenerse mediante SIMD.

3.2. BFS

Como se observa en el capítulo 2 apartado 2, BFS (Breadth-First Search) es un algoritmo utilizado para recorrer grafos de manera iterativa, explorando los nodos nivel por nivel. Presenta accesos irregulares a memoria y estructuras de datos dinámicas, lo que lo hace desafiante desde el punto de vista de la paralelización.

En cuanto a nivel de jerarquía de memoria, el CPI y el tiempo de ejecución se observa en BFS:

- En la L1, el porcentaje de tiempo consumido es muy bajo; BFS presenta un comportamiento ascendente hasta un 4 %, bfsGCC alcanza un máximo del 4 % en 4 hilos y desciende hasta un 0,5 %, y bfsAVX tiene el mejor rendimiento, con un mínimo del 0 %. En cuanto a la tasa de fallos de la L1, está por debajo del 1,75 %, por lo que es despreciable.
- En el segundo nivel de caché, el porcentaje de bloqueo por accesos es inferior al 0,7 % y la tasa de fallos oscila entre el 70 % y el 100 %, pero, al tratarse de un número tan bajo de accesos, también es despreciable.
- La L3 también es poco relevante en este caso, ya que no supera el 4 % de consumo de tiempo en accesos en ninguna de las versiones. Su tasa de fallos también es baja, con un pico del 33 % en 4 hilos para la versión bfsGCC; el resto tienen una tasa del 0 %.
- Al igual que el resto de la jerarquía de memoria, la DRAM tiene un porcentaje muy bajo de tiempo perdido en accesos, por debajo del 7%, excepto para bfs en el caso de 8 hilos que llega al 9%. La gráfica es descendente hasta llegar a menos del 4% para todas las versiones.
- En la figura 11 se puede observar que bfsAVX y bfsGCC tienen un mejor tiempo de ejecución que la versión sin vectorizar, excepto para 8 hilos. La tendencia de las gráficas es a estabilizarse a partir de los 16 hilos.
- Por último, en la figura 12 se puede observar cómo bfsAVX y bfsGCC para pocos hilos tienen un CPI medio más bajo que la versión sin vectorizar, pero van aumentando al subir el número de hilos. El CPI es óptimo, ya que se sitúa por debajo de 0,246.

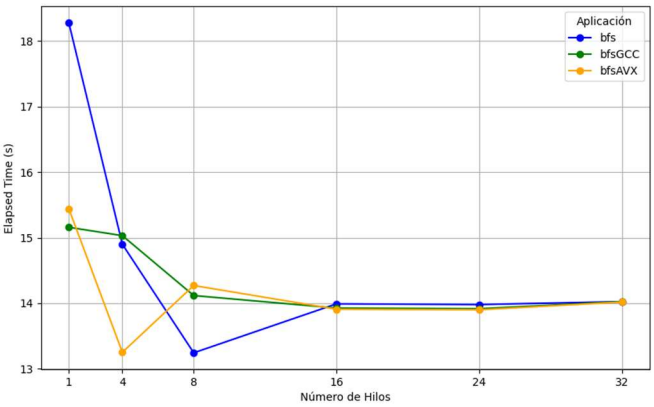

Figura 11: Gráfica que muestra una comparativa del tiempo de ejecución de BFS. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

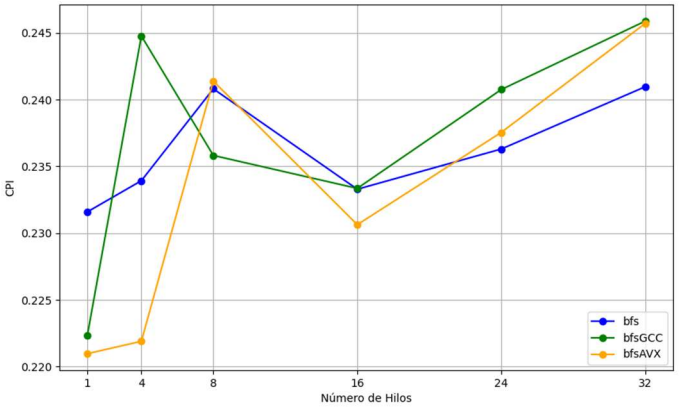

Figura 12: Gráfica que muestra una comparativa de los ciclos por instrucción que se ejecutan en BFS. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

En resumen, BFS presenta un comportamiento más favorable de lo esperado en los casos vectorizados para un patrón como Graph Traversal, donde tradicionalmente la vectorización tiene un impacto limitado debido a los accesos irregulares a memoria. En este caso, sin embargo, las versiones vectorizadas muestran mejoras claras en configuraciones con pocos hilos, siendo bfsAVX la que mejor rendimiento muestra, tanto en tiempo de ejecución como en CPI medio. Esto sugiere que, aunque el acceso a datos no sea completamente secuencial, el bajo uso de la jerarquía de memoria permite que las instrucciones SIMD se ejecuten con eficiencia en determinadas fases del algoritmo.

El hecho de que la jerarquía de memoria no represente un cuello de botella permite que el rendimiento se vea más influido por el paralelismo aplicado, aunque la escalabilidad se estabiliza a partir de los 16 hilos. Esta combinación de bajo coste en memoria y vectorización efectiva en configuraciones pequeñas convierte a BFS en un ejemplo interesante dentro del conjunto de benchmarks, donde la SIMD aporta mejoras a pesar del tipo de patrón al que pertenece.

3.3. CFD

CFD (Computational Fluid Dynamics) es un benchmark que simula el comportamiento de fluidos resolviendo numéricamente ecuaciones diferenciales sobre una malla. Requiere un elevado volumen de cómputo y trabaja con estructuras de datos no estructuradas.

A nivel de jerarquía de memoria, CPI y tiempo de ejecución se observa lo siguiente para CFD:

- En la caché de nivel uno, se observa un comportamiento ascendente en el porcentaje de tiempo que bloquea los accesos, del 1% al 4,5% para todos los ejecutables, excepto para euler3d_cpuAVX, que va del 2% al 7%. Esto implica que la versión vectorizada con `-march=sapphirerapids` accede más a L1. En cuanto a la tasa de fallos, euler3d_cpuAVX es la que más falla, pero, como su porcentaje es inferior al 5%, no es muy relevante.
- En el segundo nivel de caché, nos encontramos un bound de máximo un 0,7% por parte de euler3d_cpuAVX, el resto de ejecutables se sitúan en torno al 0,3%. El porcentaje de tasa de fallos es inferior al 6%. Esto hace que la L2 no sea muy relevante en este análisis.
- En la L3, el porcentaje de tiempo consumido por bloqueos es inferior al 4%, que asciende desde el 2% en 1 hilo para todas las versiones. En cuanto a la tasa de fallos, es del 40% hablando en porcentaje, pero que al bloquear solo el 4% del tiempo total de la ejecución, no es muy relevante.
- Como hay muy poco uso de la jerarquía de caché, a la DRAM tampoco se accede mucho, con un máximo de un 5% del tiempo bloqueando la ejecución.
- En la figura 13 se observa cómo se reduce el tiempo de ejecución hasta los 24 hilos, donde tiende a estabilizarse. También se puede observar que la versión euler3d_cpuAVX reduce el tiempo de ejecución para 1 y 4 hilos en 20 y 3 segundos, respectivamente, en comparación con el resto de las implementaciones, lo que sugiere una mejor vectorización.
- Por último, en la figura 14, que muestra el CPI medio, se puede observar un valor superior para euler3d_cpuAVX, ya que, es la que mejor vectoriza. Sin embargo, esto implica el uso de instrucciones que tardan más en ejecutarse, ya que las instrucciones SIMD trabajan con más datos.

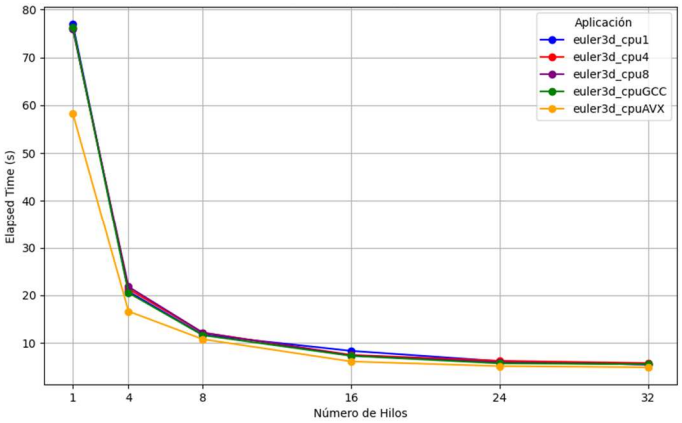

Figura 13: Gráfica que muestra una comparativa del tiempo de ejecución de CFD. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

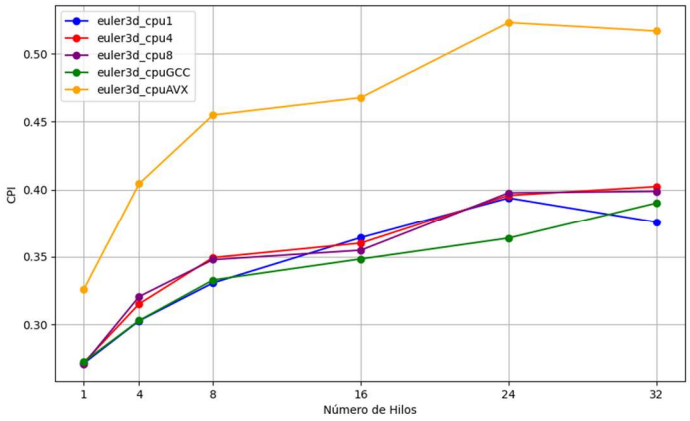

Figura 14: Gráfica que muestra una comparativa de los ciclos por instrucción que se ejecutan en CFD. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

El análisis muestra que la vectorización euler3d_cpu1GCC no aporta mejoras de rendimiento con respecto a la versión sin vectorizar, probablemente porque no logra adaptarse bien a la estructura de acceso de datos del benchmark. Sin embargo, la versión euler3d_cpuAVX, vectorizada automáticamente utilizando `-march=sapphirerapids` que está especializada en la arquitectura, sí muestra una ligera mejora en tiempo de ejecución, especialmente en configuraciones con pocos hilos. Esto indica que, para patrones no estructurados, la vectorización solo resulta efectiva cuando se adapta explícitamente a las características del hardware y del patrón de acceso.

En conjunto, CFD pone de manifiesto que la vectorización en patrones Unstructured Grid exige una adaptación específica al hardware para ser realmente efectiva. Mientras que las versiones sin vectorización o con vectorización automática simple, por parte del compilador, no muestran mejoras destacables. La versión optimizada con `-march=sapphirerapids` sí consigue aprovechar la arquitectura y superar en rendimiento al resto, especialmente en configuraciones con bajo paralelismo.

3.4. Convolution 3D

En el capítulo 2 apartado 2 se menciona que, Convolution 3D realiza operaciones de convolución sobre un volumen tridimensional de datos, aplicando filtros espaciales que combinan valores vecinos en múltiples dimensiones.

A nivel de jerarquía de memoria, CPI y tiempo de ejecución, se observa lo siguiente para Convolution 3D:

- En la caché de nivel uno, como se observa en la gráfica de la figura 15, convolution-3dAVX es la que más tiempo consume en accesos a L1, seguida de la versión sin vectorizar. El mejor porcentaje lo tiene es convolution-3dGCC. Como se indica en la figura 16, las versiones vectorizadas tienen una mayor tasa de fallos, debido a que el uso de registros SIMD puede aumentar la presión sobre la caché al procesar más datos simultáneamente.
- En el segundo nivel de caché, el porcentaje de uso del tiempo es del 7% para convolution-3dAVX, del 3,5% para convolution-3dGCC y del 1,5% para convolution-3d_acc, lo que implica una baja presión en la L2. La tasa de fallos es casi nula.
- En la L3, el comportamiento es similar al de la L2, con un porcentaje bajo de pérdida de tiempo en accesos, con una forma similar. La tasa de fallos es de 0 o 1.
- Como la L3 falla muy poco, los accesos a DRAM son muy bajos, con un comportamiento similar al de la L2 y la L3, y se mantiene el orden de consumo de tiempo.
- En la figura 17, se observa una reducción del tiempo de ejecución de Convolution 3D para 1 hilo de 10 segundos en el caso de convolution-3dGCC y de 11 segundos para convolution-3dAVX. Para el resto de los hilos, la mejora es observable con tendencia a aplanarse a partir de los 8 hilos.
- Por último, en la figura 18 se observa como convolution-3dGCC tiene el mejor rendimiento, seguido de la versión sin vectorizar y que convolution-3dAVX muestra un CPI medio entre 2,5 y 3. A pesar de tener un CPI más alto que el resto, convolution-3dAVX al utilizar instrucciones vectoriales y aprovechar mejor la jerarquía de memoria permite un tiempo de ejecución más bajo que el resto.

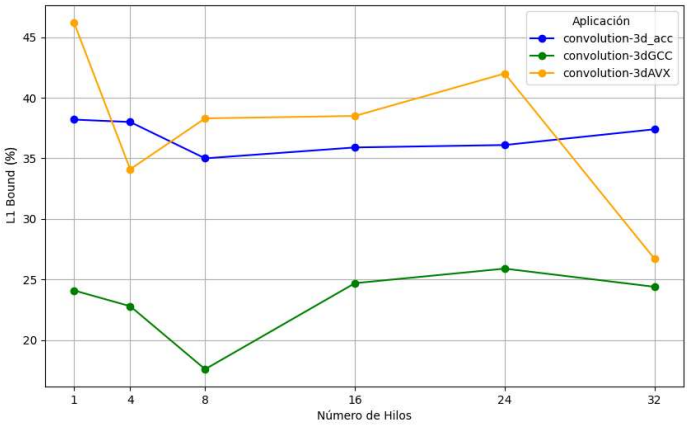

Figura 15: Gráfica que muestra una comparativa del porcentaje de tiempo que bloquea L1 la ejecución de Convolution 3D. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

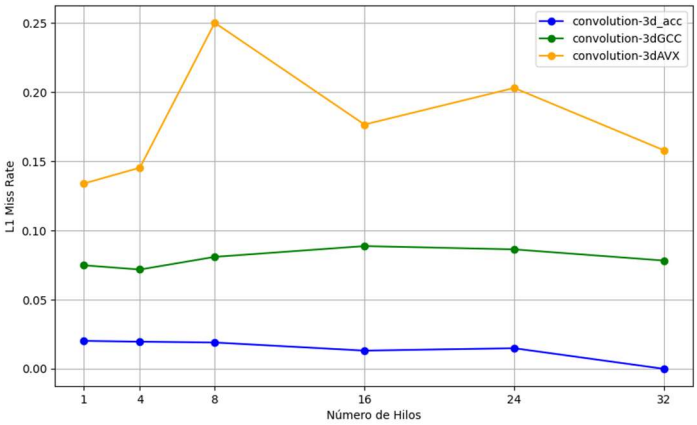

Figura 16: Gráfica que muestra una comparativa de la tasa de fallos en L3 de Convolution 3D. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

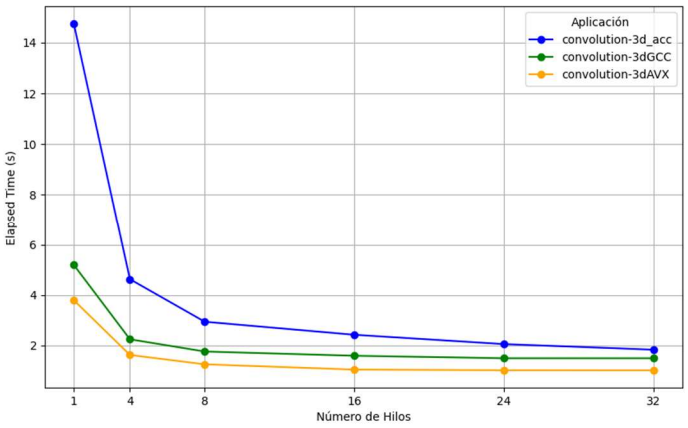

Figura 17: Gráfica que muestra una comparativa del tiempo de ejecución de Convolution 3D. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

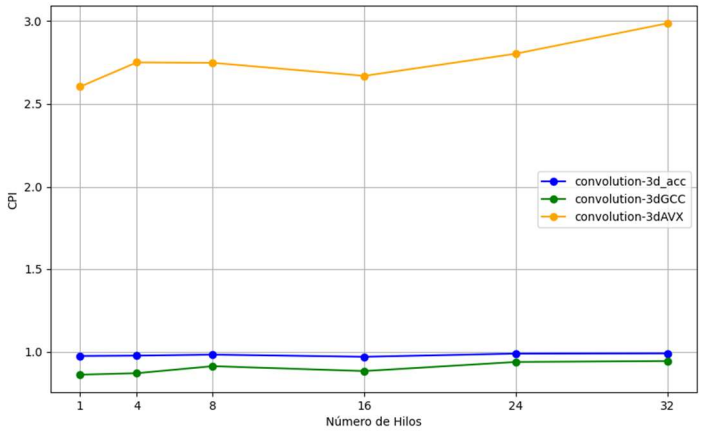

Figura 18: Gráfica que muestra una comparativa de los ciclos por instrucción que se ejecutan en Convolution 3D. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

La versión convolution-3dAVX, a pesar de tener un CPI considerablemente más alto (entre 2, 5 y 3), logra los mejores tiempos de ejecución, lo que indica que está haciendo un mejor uso de la CPU gracias a la explotación efectiva del paralelismo de datos con SIMD. Por otro lado, convolution-3dGCC ofrece un equilibrio más estable entre eficiencia de ejecución y rendimiento, con un CPI más bajo y un tiempo de ejecución ligeramente superior a convolution-3dAVX. Esto sugiere que la vectorización automática del compilador de GCC es capaz de generar código optimizado que aprovecha bien la arquitectura.

A nivel de jerarquía de memoria, el consumo de tiempo se concentra principalmente en L1, lo cual es coherente con el patrón Structured Grid y confirma que los datos accedidos entran bien en los primeros niveles de caché. La tasa de fallos en L1, alrededor del 20 % en los peores casos para las versiones vectorizadas, no representa un cuello de botella crítico.

En resumen, Convolution 3D es un benchmark que encaja perfectamente en el patrón Structured Grid, lo que implica un acceso regular a memoria y una alta reutilización espacial y temporal de datos. En conjunto, la vectorización resulta especialmente eficaz para Convolution 3D, ya que sus patrones de acceso a memoria y estructura computacional permiten explotar al máximo las instrucciones SIMD. La mejora en tiempo de ejecución es evidente, y tanto la versión convolution-3dAVX como convolution-3dGCC muestran un rendimiento notablemente superior a la versión base.

3.5. GEMM

Como se definió en el capítulo 2 apartado 2, GEMM (General Matrix Multiply) es un algoritmo clásico de álgebra lineal que realiza multiplicaciones de matrices densas. Es intensivo en operaciones de coma flotante y tiene un acceso regular a memoria.

Para GEMM a nivel de jerarquía de memoria, CPI y tiempo de ejecución se observa:

- En la L1, el porcentaje de tiempo consumido es muy bajo, por debajo del 5 %, con una tendencia media del 1 %. A pesar de esto, la mayoría de los accesos son fallidos, ya que oscilan entre el 60 % y el 69 %, porcentaje que aumenta al aumentar el número de hilos.
- En el segundo nivel de caché, hay tanto poco porcentaje del tiempo perdido en accesos, que presentan entre el 1% y el 7% que asciende con el número de hilos, como poca tasa de fallos, inferior al 6% descendente hasta un 4,5% en 32 hilos.
- En la gráfica de la figura 19, se puede observar como para la L3, en este caso hay una tendencia descendente, donde gemm_acc es la que muestra peor comportamiento y gemmGCC, la que mejor para casos intermedios. gemmAVX es la que mejor rendimiento presenta en accesos a L3 en casos extremos, es decir, 1, 24 y 32 hilos. La tasa de fallos de L3 es muy baja siendo casi del 0%.
- Como L3 falla muy poco, los accesos a DRAM son muy bajos, y consumen un tiempo que oscila entre el 4% y el 0%.

- En la figura 20, que muestra el tiempo ejecución de GEMM, se observa un comportamiento decreciente con poca mejora de rendimiento a partir de los 8 hilos y con unos tiempos muy similares para las 3 versiones de GEMM.
- Por último, en la figura 21, que muestra el CPI medio, se observa un buen CPI para todas las versiones, siendo gemmAVX la que tiene un CPI medio peor para todos los hilos. Las versiones gemm_acc y gemmGCC tienen un CPI medio similar.

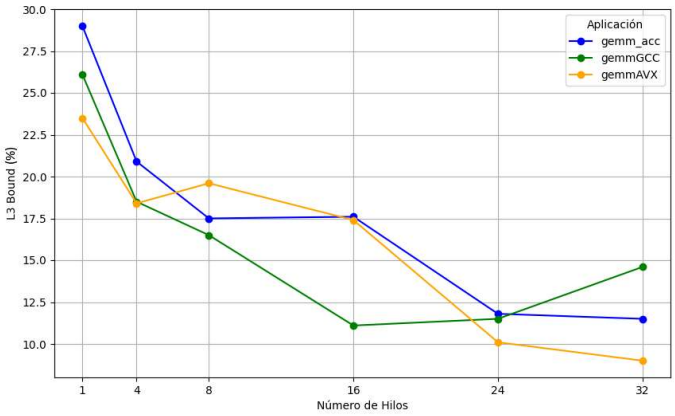

Figura 19: Gráfica que muestra una comparativa del porcentaje de tiempo que bloquea L3 la ejecución de GEMM. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

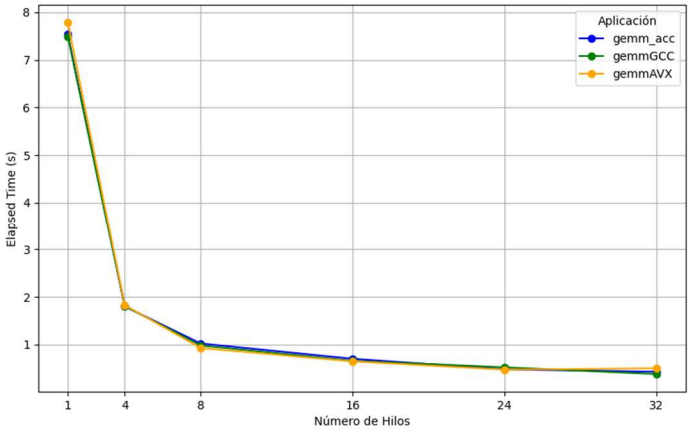

Figura 20: Gráfica que muestra una comparativa del tiempo de ejecución de GEMM. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

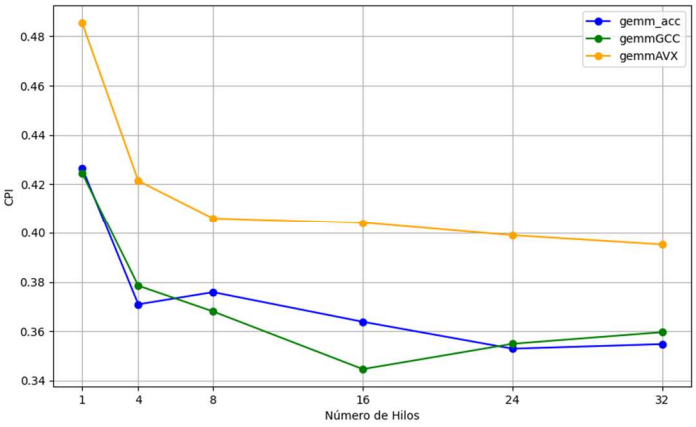

Figura 21: Gráfica que muestra una comparativa de los ciclos por instrucción que se ejecutan en GEMM. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

Desde el punto de vista del rendimiento, los tiempos de ejecución muestran una mejora progresiva hasta 8 hilos, pero a partir de ahí la ganancia es muy reducida, lo que sugiere una saturación del paralelismo a nivel de hilos. Además, las diferencias entre versiones vectorizadas y no vectorizadas son mínimas en tiempo de ejecución, con diferencias de apenas unos milisegundos. En cuanto al CPI, se mantienen valores bajos en general, indicando buena eficiencia computacional. La versión gemmAVX presenta un CPI ligeramente más alto en comparación con las otras dos, lo que podría explicarse por el coste de manejar registros vectoriales más anchos sin una ganancia proporcional en rendimiento.

En resumen, GEMM es un benchmark altamente computacional, al usar el patrón Dense Linear Algebra, cuyo rendimiento no está limitado por la jerarquía de memoria. Sin embargo, la vectorización no aporta mejoras significativas, ya que todas las versiones muestran un comportamiento muy similar. Esto sugiere que la operación ya está suficientemente optimizada y que las técnicas SIMD no logran incrementar de forma sustancial el rendimiento debido a una saturación temprana del paralelismo disponible.

3.6. Hotspot

Hotspot es un benchmark que modela la distribución de temperatura en una superficie a lo largo del tiempo, utilizando un esquema iterativo basado en diferencias finitas. también, es importante recordar que como se menciona en el capítulo 2 apartado 5, Hotspot es un benchmark que se pudo vectorizar manualmente por lo que tiene 2 versiones más.

Para Hotspot a nivel de jerarquía de memoria, CPI y tiempo de ejecución se observa:

- Como se muestra en la figura 22, en la L1 hay un porcentaje de consumo de tiempo diferenciado en 2 grupos. Por una parte, está hotspotAVX y, por otra, el resto de las versiones. HotspotAVX mantiene un consumo del 16%, mientras que el resto muestran un comportamiento creciente, siendo hotspotGCC la que menor consumo de tiempo presenta. Al observar la figura 23 sobre la tasa de fallos de L1, se puede apreciar claramente la diferencia entre versiones, lo que concuerda aproximadamente con la gráfica anterior.
- En el segundo nivel de caché, pasa algo parecido a lo que ocurre en la L1. Con un máximo de 13% en 1 hilo y un mínimo de 5% en 32 hilos, hotspotAVX muestra un comportamiento peor que el resto de las versiones. Por el contrario, hotspotGCC tiene un porcentaje rondando el 1%, con el resto de

las versiones con un 0,5% más de consumo temporal. En este caso, la tasa de fallos es muy baja, por debajo del 0,8%.

- En el caso de la L3, el consumo de tiempo de ejecución en accesos es inferior al 8 % en todos los casos y todas las versiones muestran una forma muy similar, aunque hotspotGCC tiene un mejor comportamiento. La tasa de fallos en la L3 es del 0 % o del 100 %, y los que la tienen son: hotspotAVX, hotspot4 y hotspot8 con 8 hilos, hotspot1 con 24 hilos y hotspotGCC con 32 hilos.
- En el caso de DRAM bound, todas tienen un porcentaje inferior al 3%, excepto hotspotAVX, que tiene un pico del 5,5%. Todas las versiones se comportan de una forma similar, menos hotspotGCC que tiene un porcentaje ligeramente inferior.
- En la figura 24 de tiempo ejecución de Hotspot, se observa un comportamiento de descenso brusco de 1 a 4 hilos, seguido de un descenso menos pronunciado a 8 hilos y una mejora menos significativa a partir de 16 hilos. La que peores tiempos tiene es hotspotAVX, y el resto tienen unos tiempos muy parecidos, con diferencias de pocos segundos.
- Por último, en la figura 25 se puede observar que todas las versiones de Hotspot tienen un CPI medio muy bajo, donde hotspotGCC tiene el menor CPI de todos y hotspotAVX el mayor, con un máximo cercano a 0,9.

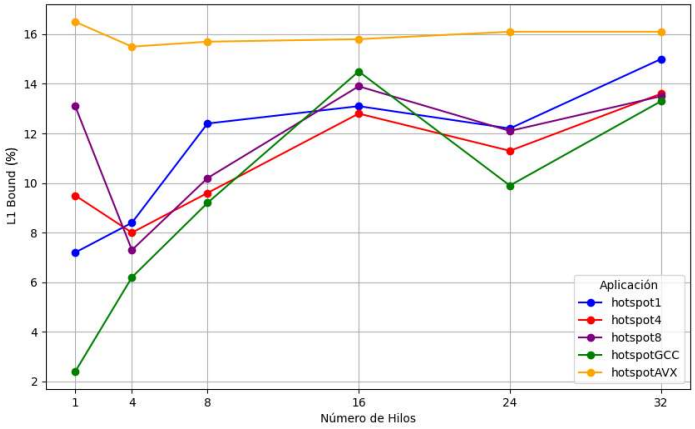

Figura 22: Gráfica que muestra una comparativa del porcentaje de tiempo que bloquea L1 la ejecución de Hotspot. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

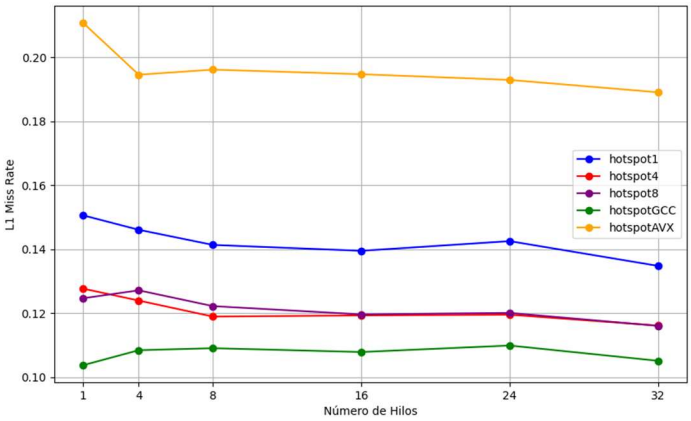

Figura 23: Gráfica que muestra una comparativa de la tasa de fallos en L1 de Hotspot. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

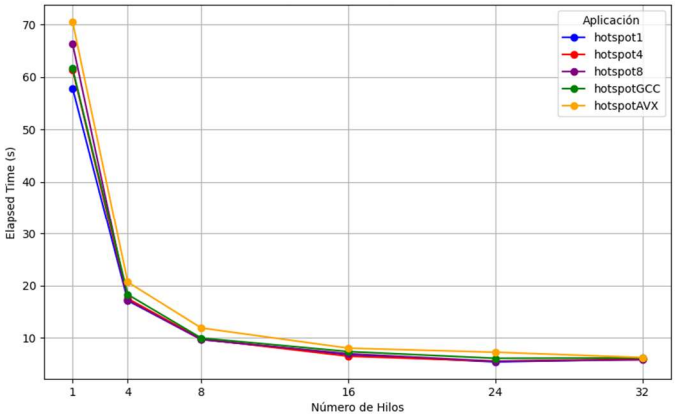

Figura 24: Gráfica que muestra una comparativa del tiempo de ejecución de Hotspot. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

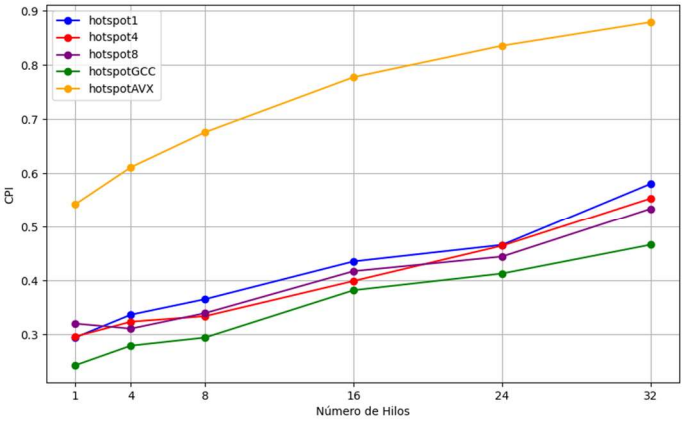

Figura 25: Gráfica que muestra una comparativa de los ciclos por instrucción que se ejecutan en Hotspot. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

En resumen, Hotspot muestra un comportamiento coherente con su patrón Structured Grid, con un acceso regular a memoria y un uso razonablemente eficiente de la jerarquía de caché. Sin embargo, las diferencias entre versiones son notables. hotspotAVX presenta sistemáticamente un rendimiento inferior, tanto en consumo de memoria como en tiempo de ejecución y CPI, lo que indica que la vectorización automática con `-march=sapphirerapids` no se adapta bien a las características del benchmark. Por el contrario, hotspotGCC, que emplea vectorización automática por parte del compilador, alcanza los mejores resultados globales, con un CPI más bajo, menor uso de caché y mejores tiempos de ejecución.

Estas observaciones sugieren que, en este caso, la vectorización de hotspotAVX no solo no mejora el rendimiento, sino que puede ser contraproducente, posiblemente debido al sobrecoste de manipular registros vectoriales más anchos o a una mala alineación con el patrón de acceso a datos. Por otro lado, la versión hotspotGCC muestra un mejor comportamiento, pero no lo suficientemente remarcable como para que sea una mejora sustancial del rendimiento del benchmark. Por lo tanto, la estrategia de vectorización elegida resulta clave para obtener mejoras reales en benchmarks con acceso a memoria estructurado, como Hotspot.

3.7. Jacobi 2D

Como se definió en el capítulo 2 apartado 2, Jacobi 2D es un algoritmo iterativo utilizado para resolver sistemas de ecuaciones lineales, aplicando una técnica de relajación sobre una malla bidimensional.

A nivel de jerarquía de memoria, el CPI y el tiempo de ejecución se observa lo siguiente para Jacobi 2D:

- En la L1, el tiempo de ejecución aumenta progresivamente, y las tres versiones de Jacobi 2D empiezan con un 8 % y terminan con un 15 %. En cuanto al porcentaje de tasa de fallos de la L1, la versión AVX tiene una media del 40 %, la vectorizada por GCC del 25 % y la versión sin vectorizar del 7 %. Esto supone que, aunque el porcentaje de tiempo que la L1 bloquea la ejecución no es muy elevado, la vectorización aumenta los fallos en los accesos.
- En el segundo nivel de caché, en este caso el bloqueo proveniente de los accesos supone menos del 2,25% del total. Y la tasa de fallos es muy parecida a la de L1, solamente que en este caso es un 10% menor en las 3 versiones.
- En la figura 26 de L3, se observa como jacobi-2d-imperAVX tiene forma de arco debido a que al ir aumentando el número de hilos hasta 16, hace que la vectorización pierda efectividad y genere más accesos. La bajada se debe a que el aumento de hilos compensa la vectorización. Por otro lado, las otras dos versiones presentan un comportamiento ascendente. El porcentaje de la tasa de fallos de L3 son en todas las versiones del 100%, debido a que los datos no entran el L3.
- Como se muestra en la figura 27 correspondiente al porcentaje de bloqueos por accesos a DRAM, se puede observar como la que más se bloquea es jacobi-2d-imperAVX, seguida de jacobi-2d-imperGCC y, por último, jacobi-2d-imper_acc. Como se ha podido observar, las versiones vectorizadas son las que más presión ejercen en toda la jerarquía de memoria.
- En la figura 28 se puede observar como el tiempo de ejecución es parejo para todas las versiones con una bajada pronunciada de 1 a 4 hilos, menos a partir de 8 hilos donde hay una diferencia de 1 segundo entre versiones, aunque la tendencia es a estabilizarse.
- Por último, en la figura 29 se puede observar cómo jacobi-2d-imperAVX es el que mayor CPI medio tiene, seguido de jacobi-2d-imperGCC y por último la versión sin vectorizar. Esto se debe a que las instrucciones vectoriales tardan más ciclos en ejecutarse y ponen más presión sobre la jerarquía de memoria.

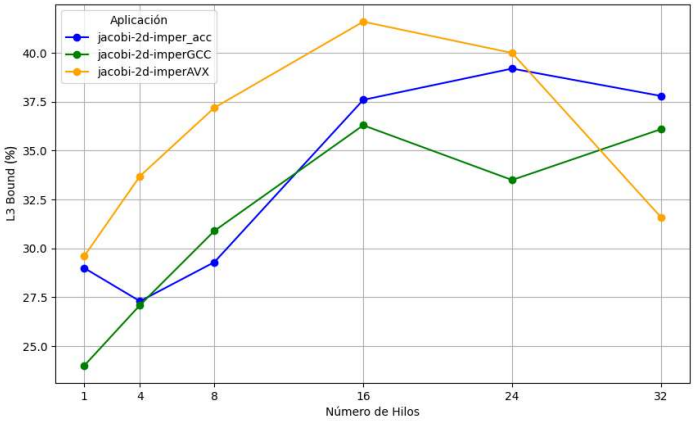

Figura 26: Gráfica que muestra una comparativa del porcentaje de tiempo que bloquea L3 la ejecución de Jacobi 2D. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

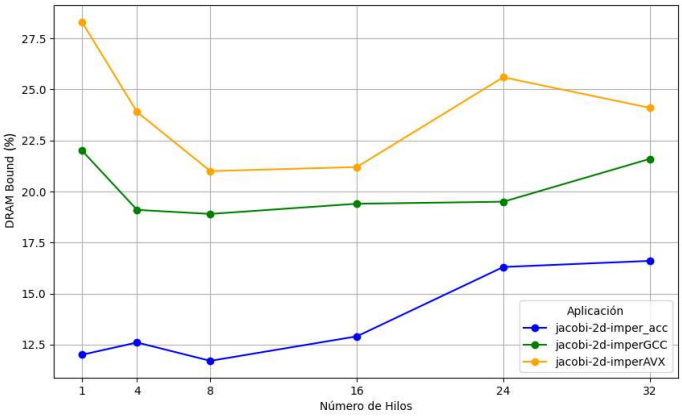

Figura 27: Gráfica que muestra una comparativa del porcentaje de tiempo que bloquea DRAM la ejecución de Jacobi 2D. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

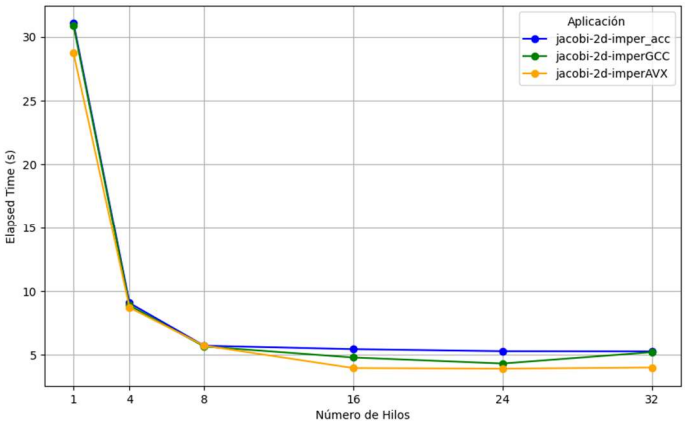

Figura 28: Gráfica que muestra una comparativa del tiempo de ejecución de Jacobi 2D. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

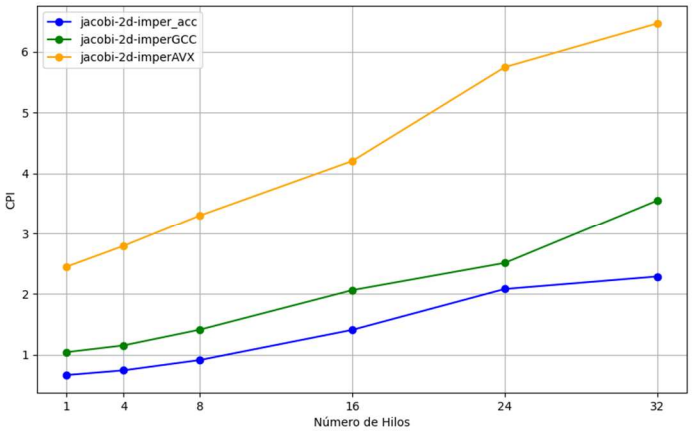

Figura 29: Gráfica que muestra una comparativa de los ciclos por instrucción que se ejecutan en Jacobi 2D. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

En este caso, los resultados muestran que la jerarquía de memoria constituye un cuello de botella importante, especialmente en la caché L3 y en el acceso a memoria principal. La tasa de fallos en L3 alcanza el 100 % en todas las versiones, lo que indica que los datos no consiguen mantenerse en ese nivel, forzando accesos constantes a DRAM. Este efecto se acentúa en las versiones vectorizadas, en particular en jacobi-2d-imperAVX, que presenta los peores valores en DRAM Bound y en consumo de tiempo bloqueado por accesos a memoria.

No obstante, y a pesar de tener un CPI notablemente más alto, las versiones vectorizadas obtienen mejores tiempos de ejecución a partir de 8 hilos. Esto se debe a que el mayor volumen de trabajo procesado por instrucción SIMD compensa el coste adicional en ciclos y la presión sobre la memoria, permitiendo completar el cálculo en menos tiempo real a pesar de una menor eficiencia por instrucción.

Este comportamiento pone de relieve una situación interesante: aunque la eficiencia interna se reduce con la vectorización, el rendimiento mejora gracias a una mayor explotación del paralelismo de datos. Es decir, se sacrifica eficiencia para ganar velocidad total, algo común en algoritmos donde el cómputo domina sobre el acceso a memoria. En conjunto, Jacobi 2D demuestra que la vectorización puede ser efectiva incluso cuando la jerarquía de memoria actúa como limitante, siempre que se acompañe de un grado suficiente de paralelismo. La elección entre `–march=sapphirerapids` y `–ftree-vectorize` depende de si se prioriza la reducción del tiempo total o la eficiencia interna del sistema.

3.8. LavaMD

Como se definió en el capítulo 2 apartado 2, LavaMD simula las interacciones moleculares entre partículas en un espacio tridimensional, calculando fuerzas en función de distancias entre pares de elementos.

Para LavaMD a nivel de jerarquía de memoria, CPI y tiempo de ejecución se observa:

- En la L1 hay un porcentaje de consumo de tiempo en del 0,6% para todas las versiones, una cantidad muy baja, posiblemente debida a un bajo acceso a memoria y una carga de trabajo computacional elevada. El porcentaje de tasa de fallos de L1 es inferior a 1% por lo que es despreciable.
- En el segundo nivel de caché, en este caso el bloqueo proveniente de los accesos supone menos del 0.3%, al igual que L1 debido a un número bajo de accesos a memoria. Los fallos de L2 son de 0% excepto para LavaMD en 8 hilos que ascienden a un 100% del total. Este pico se debe a una llamada al sistema.
- En la L3, el comportamiento es igual que en la L2, tanto en fallos como porcentaje del tiempo debido a bloqueos. Las razones son las mismas, bajos accesos a memoria y fallos debidos solamente a llamadas al sistema.
- El DRAM Bound es de 0% en toda la extensión y para todas las versiones. Los accesos a DRAM producidos por la llamada al sistema no se hacen presentes en los resultados.

- En la figura 30, se puede observar que las versiones vectorizadas presentan un tiempo de ejecución más bajo, de unos 30 segundos menos para 1 hilo y 40 segundos menos para 4 hilos. A partir de 8 hilos, las gráficas convergen. Esto se debe a que la vectorización está siendo efectiva en los casos en los que la paralelización es baja.
- Por último, en la figura 31 se puede observar que el CPI medio de las 3 versiones de LavaMD son muy parecidas, variando en una escala que no presenta mucha diferencia.

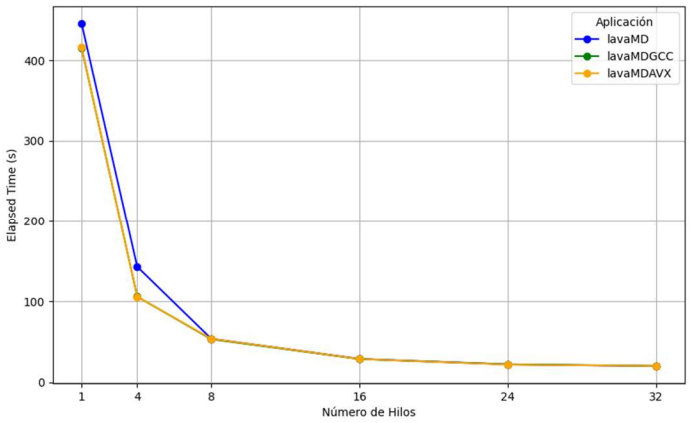

Figura 30: Gráfica que muestra una comparativa del tiempo de ejecución de LavaMD. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

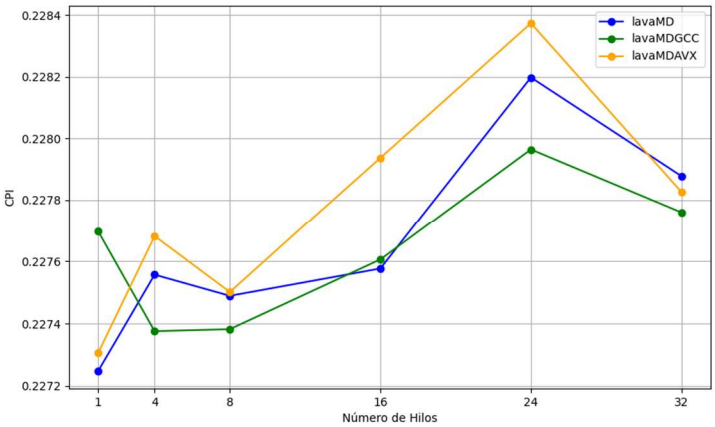

Figura 31: Gráfica que muestra una comparativa de los ciclos por instrucción que se ejecutan en LavaMD. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

En resumen, LavaMD representa un claro ejemplo del patrón computacional N-Body de los Berkeley Dwarfs, caracterizado por la simulación de interacciones entre partículas mediante cálculos intensivos basados en distancias y fuerzas. Este patrón suele presentar un alto coste computacional con accesos a memoria relativamente reducidos, lo cual se refleja perfectamente en los resultados obtenidos.

El tiempo de ejecución disminuye drásticamente al aumentar el número de hilos y la vectorización resulta especialmente efectiva en configuraciones con bajo paralelismo. Esto se debe a que, en ausencia de cuellos de botella en memoria, el paralelismo de datos que ofrece SIMD permite aprovechar al máximo los recursos computacionales del sistema en niveles bajos de concurrencia y a niveles altos la paralelización tiene una alta escalabilidad.

En conjunto, LavaMD es un benchmark dominado por el cálculo, que se ajusta bien al patrón N-Body y escala de forma excelente tanto con hilos como con vectorización, convirtiéndose en un caso ideal para analizar la eficiencia de paralelismo en ausencia de presión sobre la jerarquía de memoria.

3.9. SpMV

Como se definió en el capítulo 2 apartado 2, SpMV (Sparse Matrix-Vector Multiply) realiza la multiplicación entre una matriz dispersa y un vector, operación común en simulaciones científicas y álgebra lineal escasa.

Para SpMV a nivel de jerarquía de memoria, CPI y tiempo de ejecución se observa:

- En la L1 hay un porcentaje de consumo de tiempo con un pico de un 45% en 2 hilos y un pico del 85% en 32 hilos, todas las versiones de SpMV comparten el mismo porcentaje. En cuanto a la tasa de fallos se observa que spmvAVX tiene un porcentaje de tasa de fallos del 37% y las otras 2 versiones del 23%.
- En el segundo nivel de caché, el porcentaje de consumo de tiempo en accesos es inferior al 4%, menos para spmvAVX en 1 hilo, que llega al 10%. La tasa de fallos para spmvAVX es en porcenta un 23% y para el resto menos del 5%. La L2 en este benchmark, no genera mucho cuello de botella.
- Como se observa en la gráfica de la figura 32, todas las versiones tienen el mismo porcentaje de consumo de tiempo por accesos a L3, llegando a ser de un 87% en 24 hilos. La tasa de fallos por otro lado tiene un comportamiento contrario, en 2 hilos falla el 87% de los accesos y se reduce hasta 0% en 32 hilos. En este caso, la versión spmvAVX tiene una tasa de fallos inferior al resto, que alcanza un máximo del 70 %.
- En la figura 33 de DRAM bound se puede observar como todas las gráficas tiene un comportamiento similar llegando a su máximo de consumo de tiempo de ejecución en 4 hilos con un 15%.
- En la figura 34, se observa que todas las versiones tienen un tiempo de ejecución muy parecido, y que al paralelizar, los tiempos pasan de estar en torno a los 60 segundos a estar en torno a los 180. Cuando van aumentando los hilos mejora el tiempo de ejecución, pero la mejor versión es la de 1 hilo.
- Por último, en la figura 35 el CPI medio muestra unos valores muy altos, debido sobre todo a la jerarquía de memoria, sobre todo a la L3. La versión de spmvAVX es la que tiene valores más altos y el resto de las versiones mantienen un CPI muy similar.

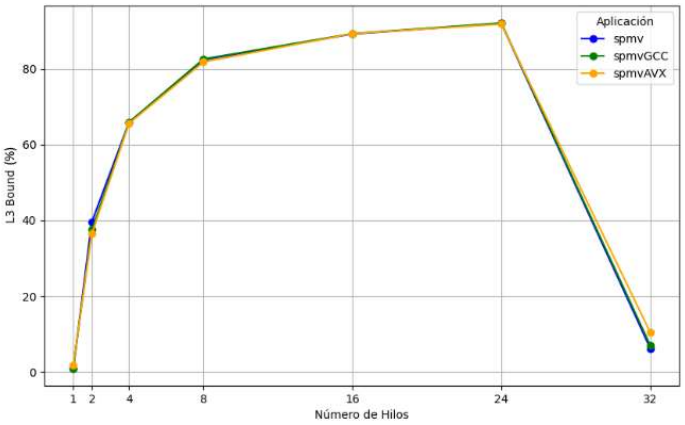

Figura 32: Gráfica que muestra una comparativa del porcentaje de tiempo que bloquea L3 la ejecución de SpMV. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

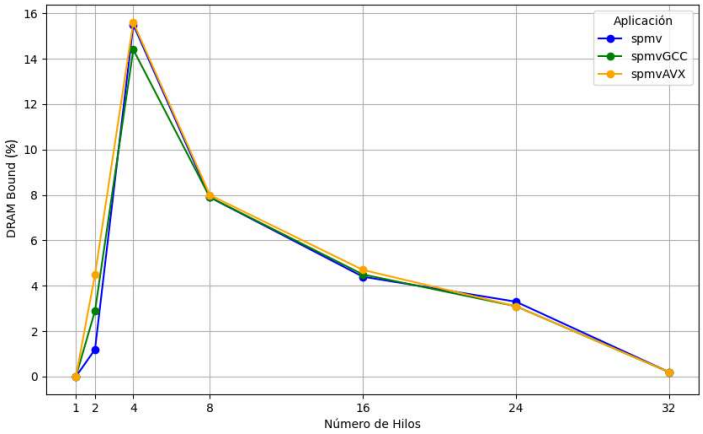

Figura 33: Gráfica que muestra una comparativa del porcentaje de tiempo que bloquea DRAM la ejecución de SpMV. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

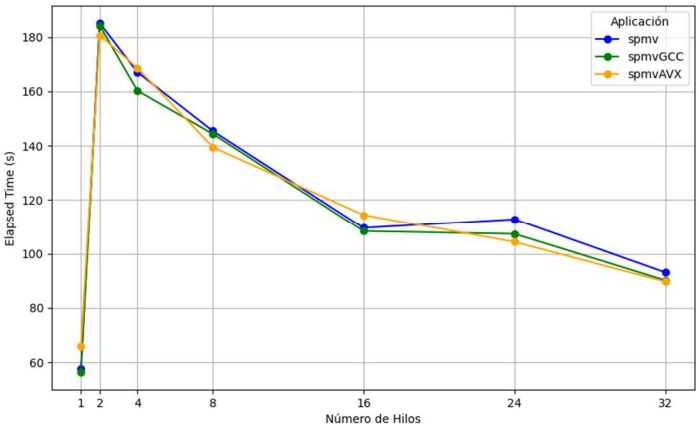

Figura 34: Gráfica que muestra una comparativa del tiempo de ejecución de SpMV. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

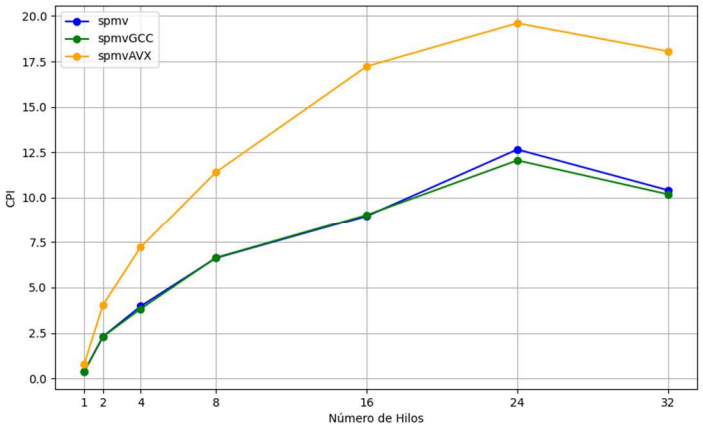

Figura 35: Gráfica que muestra una comparativa de los ciclos por instrucción que se ejecutan en SpMV. Para cada una de las implementaciones se muestra el avance de dicho porcentaje en referencia al número de hilos que se han empleado: 1, 4, 8, 16 y 32.

En resumen, SpMV pertenece al patrón Sparse Linear Algebra, caracterizado por accesos irregulares a memoria y baja densidad de cómputo. Este patrón complica tanto la paralelización como la vectorización, algo que se refleja en los resultados: la versión con 1 hilo presenta el mejor rendimiento, y el tiempo de ejecución empeora inicialmente al paralelizar, lo que motivó la inclusión de un punto intermedio con 2 hilos.

El principal cuello de botella es la jerarquía de memoria, especialmente la caché L3, que llega a bloquear hasta el 87 % del tiempo de ejecución en ciertas configuraciones. Aunque la versión spmvAVX tiene una tasa de fallos algo menor, no aporta mejoras significativas en rendimiento, y todas las versiones muestran un CPI alto, indicando que la CPU pasa mucho tiempo esperando a memoria.

En conjunto, SpMV demuestra las limitaciones de las optimizaciones tradicionales cuando el acceso irregular a memoria domina la ejecución, y pone en evidencia que ni la vectorización ni el paralelismo por hilos logran escalar eficazmente en este tipo de aplicaciones.

4. Conclusiones y trabajo futuro

En este trabajo se ha analizado el comportamiento de nueve benchmarks pertenecientes a distintos patrones computacionales (Berkeley Dwarfs), lo que ha permitido evaluar la influencia del paralelismo por hilos, la vectorización (tanto automática como mediante OpenMP), y el uso de la jerarquía de memoria en el rendimiento global de cada aplicación.

Los resultados muestran que los benchmarks con patrones computacionales regulares, como Backprop, BFS, Jacobi 2D o Convolution 3D, se benefician claramente de la vectorización y de la paralelización por hilos, aunque en algunos casos el rendimiento se ve limitado por el acceso a niveles específicos de memoria, como L1 o L3. En estos casos, la versión vectorizada automáticamente con el flag `–march=sapphirerapids`, tiende a ofrecer un mejor rendimiento cuando está bien alineada con la arquitectura, aunque no siempre reduce el CPI.

Por otro lado, los benchmarks como GEMM y Hotspot, a pesar de pertenecer a patrones computacionales regulares (Dense Linear Algebra y Structured Grid, respectivamente), no muestran una mejora significativa al aplicar vectorización. En el caso de GEMM, el rendimiento entre versiones es muy similar, con diferencias de apenas milisegundos, lo que sugiere que la versión base ya está bien optimizada y que la vectorización no aporta un beneficio claro. Hotspot, por su parte, presenta un comportamiento aún más llamativo: la versión vectorizada con `–march=sapphirerapids` obtiene peores resultados en cuanto a tiempo de ejecución y CPI que las otras versiones, lo que indica que una vectorización más agresiva no siempre garantiza una mejora si no está bien alineada con el patrón de acceso a memoria de la aplicación.

En cambio, aplicaciones con patrones más irregulares, como SpMV y en menor medida CFD, no obtienen mejoras significativas mediante la vectorización automática, y la paralelización muestra una escalabilidad limitada. En estos casos, la jerarquía de memoria, especialmente L3 y DRAM, actúan como cuellos de botella dominantes que afectan negativamente al CPI y al tiempo de ejecución. En particular, SpMV demuestra los límites de la vectorización tradicional cuando el acceso a memoria es disperso.

Por otro lado, los benchmarks como LavaMD, del patrón N-Body, muestran un comportamiento opuesto: al tener una carga computacional muy alta y pocos accesos a memoria, es capaz de escalar perfectamente tanto con hilos como con vectorización. En este caso, el CPI se mantiene estable y bajo, y la jerarquía de memoria no limita en absoluto el rendimiento.

También se ha observado que la vectorización es más efectiva en configuraciones con un bajo número de hilos, donde el paralelismo de datos puede complementar la

falta de paralelismo a nivel de hilos. No obstante, a medida que se incrementa el número de hilos, los beneficios de la vectorización tienden a reducirse o incluso desaparecer, en especial cuando aparecen cuellos de botella en memoria o cuando la carga no se distribuye de manera óptima.

Además, se ha observado que, en la mayoría de los casos, a partir de ocho o dieciséis hilos, se alcanza un punto de estancamiento en la curva de rendimiento, por lo que añadir más hilos no conlleva mejoras significativas. Este comportamiento puede explicarse mediante la ley de Amdahl, que establece un límite teórico a las ganancias obtenidas mediante paralelización en función de la fracción secuencial del código. En muchos benchmarks, una vez superado cierto umbral de paralelismo, los factores limitantes pasan a ser otros, como la saturación de la jerarquía de memoria, la sobrecarga de sincronización entre hilos, o el desequilibrio en la distribución de la carga. En consecuencia, el rendimiento tiende a estabilizarse o incluso a degradarse, lo que indica que se ha alcanzado el límite de escalabilidad posible para esa aplicación concreta en la arquitectura analizada.

En conjunto, el análisis demuestra que no existe una solución única que mejore el rendimiento en todos los casos. La efectividad de las optimizaciones depende en gran medida del patrón computacional, del tipo de acceso a memoria, y de cómo se alinean las técnicas de vectorización y paralelización con la arquitectura del sistema. Por ello, conocer el patrón Berkeley Dwarf asociado a una aplicación puede ser una herramienta clave a la hora de decidir cómo optimizarla.

Este trabajo ofrece una base sólida para el análisis del rendimiento en sistemas multinúcleo, pero también sugiere varias líneas de mejora y ampliación para futuros estudios.

Una primera ampliación natural sería incorporar métricas energéticas al análisis. Tal y como se menciona en el apartado 6 del capítulo 2, se consideró inicialmente evaluar el consumo energético de las distintas implementaciones. Sin embargo, debido a limitaciones técnicas a la hora de obtener resultados con Intel VTune, no fue posible incluir estos datos en el análisis final. Si se incluyera esta métrica, se podría estudiar no solo cuál es la versión más rápida, sino también cuál es la más eficiente en términos de consumo energético, lo que resulta especialmente relevante en entornos en los que el rendimiento por vatio es un factor crítico.

Otra posible ampliación del análisis sería replicarlo en sistemas distribuidos o con arquitecturas más complejas, como clústeres con múltiples nodos de cómputo o procesadores heterogéneos. De este modo, se podría observar cómo escalan las implementaciones más allá de una única máquina, y qué nuevos cuellos de botella surgen cuando entran en juego factores como la comunicación entre nodos o la distribución de memoria.

Por último, sería muy valioso ampliar el conjunto de benchmarks utilizados, incorporando aplicaciones que representen otros patrones computacionales (Berkeley Dwarfs) no cubiertos en este estudio. Esto proporcionaría una visión más completa sobre el impacto de las técnicas de paralelización y vectorización en diferentes dominios, ayudando a generalizar mejor las conclusiones obtenidas.

5. Bibliografía

[1] Intel Corporation, *Intel® VTune™ Profiler Performance Analysis Cookbook*, 2023. [Online]. Disponible: https://cdrdv2-public.intel.com/773629/vtune-profiler_cookbook_2023.1-766316-773629.pdf

[2] OpenMP Architecture Review Board, *OpenMP Application Program Interface Version 5.0*, Nov. 2018. [Online]. Disponible: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[3] R. D. Blumofe et al., "Cilk: An Efficient Multithreaded Runtime System," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, Aug. 1996.

[4] J. S. Gruen, "Crash Course: Introduction to Parallelism – SIMD Parallelism," *Johnny's Software Lab*, 28-Aug-2021. [Online]. Disponible: https://johnnysswlab.com/crash-course-introduction-to-parallelism-simd-parallelism

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, y K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," University of Virginia. [Online]. Disponible: https://www.cs.virginia.edu/rodinia/

También disponible: https://github.com/yuhc/gpu-rodinia

[6] C. Lab, "PolyBench-ACC: Polyhedral Benchmark Suite with Accelerator Support," *GitHub*, 2020. [Online]. Disponible: https://github.com/cavazos-lab/PolyBench-ACC

[7] computablee, "heterogeneous-spmv: Benchmarking Sparse Matrix-Vector Multiplication (SpMV) across Heterogeneous Architectures," *GitHub*, 2023. [Online]. Disponible: https://github.com/computablee/heterogeneous-spmv

[8] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple Linux Utility for Resource Management," in *Proceedings of the Job Scheduling Strategies for Parallel Processing (JSSPP)*, Springer, 2003, pp. 44–60.

[9] Intel Corporation, "Intel® Xeon® Gold 6416H Processor," *Intel ARK*, [Online]. Disponible: https://www.intel.com/content/www/us/en/products/sku/232389/intel-xeon-gold-6416h-processor-45m-cache-2-20-ghz/specifications.html

[10] dgc450, "TFG_Rendimiento" GitHub, 2025. [Online]. Disponible: https://github.com/dgc450/TFG_Rendimiento/tree/main

[11] K. Asanović, R. Bodík, B. C. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. Plishker, J. Shalf, S. W. Williams y K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," EECS Department, University of

California, Berkeley, Tech. Rep. UCB/EECS-2006-183, 2006. [Online]. Disponible: https://web.stanford.edu/class/ee380/Abstracts/070131-BerkeleyView1.7.pdf