Uso de extensiones vectoriales SIMD para acelerar el alineamiento de secuencias

Cristian Iñiguez Rodriguez

2 de julio de 2024

Resumen– La alineación por pares en el análisis de secuencias genómicas es crucial y se lleva a cabo mediante algoritmos basados en programación dinámica. El estado del arte actual, WaveFront Alignment (WFA), ofrece oportunidades para aprovechar instrucciones vectoriales a nivel de máquina, como las extensiones SIMD. Este proyecto se centra en explorar alternativas de diseño e implementación para integrar instrucciones SIMD en WFA, implicando la reestructuración de las estructuras de datos y la modificación del acceso a estos datos. Se desarrollarán e investigarán implementaciones para diversas arquitecturas de conjuntos de instrucciones, como AVX2/AVX512 (Intel, AMD). La evaluación del rendimiento se llevará a cabo en plataformas de Computación de Alto Rendimiento (HPC). Como resultado final, se desarrollará una interfaz web para visualizar estadísticas de la ejecución del programa.

Palabras clave- Algoritmo de alineamiento wavefront (WFA), alineamiento de secuencias de pares, AVX, distancia Levenshtein, edit-distance, SIMD, SVE, vectorización.

Abstract– Pairwise alignment in genomic sequence analysis is crucial and is performed using dynamic programming-based algorithms. The state of art, WaveFront Alignment (WFA), offers opportunities to leverage machine-level vector instructions, such as SIMD extensions. This project focuses on exploring design and implementation alternatives for integrating SIMD instructions into WFA, involving restructuring data structures and modifying access to this data. Implementations will be developed and investigated for various instruction set architectures, such as AVX2/AVX512 (Intel, AMD). Performance evaluation will be performed on High Performance Computing (HPC) platforms. As a final result, a web interface will be developed to visualize statistics of the program execution.

Keywords– AVX, edit-distance, Levenshtein distance, pairwise sequence alignment, SIMD, SVE, vectorization, wavefront alignment algorithm (WFA).

1 Introducción

Los avances en la tecnología de secuenciación del ADN han provocado un crecimiento exponencial en la generación de datos biológicos [1] y una gran reducción en el coste para obtenerlos. Estos datos biológicos consisten en secuencias de nucleótidos (adenina, citosina, guanina y timina, comúnmente representadas por las letras A, C, G y T). Este incremento exponencial ha generado una demanda de herramientas bioinformáticas más rápidas y eficientes para mantener el ritmo de producción de datos que los científicos biomédicos utilizan para comprender la evolución, identificar genes, y predecir estructuras y funciones de biomoléculas.

Una de las tareas bioinformáticas más frecuentes es el alineamiento de secuencias. Históricamente, este problema ha sido abordado con variantes de los algoritmos Needleman-Wunsch [2] y Smith-Waterman-Gotoh [3]. En la actualidad, el estado del arte es el algoritmo de alineación Wavefront (WFA) [4], que aprovecha las similitudes entre las secuencias para reducir el trabajo computacional total, logrando así alinear secuencias mucho más rápidamente que los algoritmos clásicos. Sin embargo, las implementaciones actuales del WFA no aprovechan eficientemente el conjunto de instrucciones de los procesadores modernos, lo que sugiere que el rendimiento del algoritmo es susceptible de mejorar aún más. Este trabajo emplea metodologías de ingeniería de rendimiento para acelerar el algoritmo WFA. Se lleva a

• E-mail de contacte: 1566514@uab.cat
• Menció realitzada: Tecnologies de la Informació
• Treball tutoritzat per: Juan Carlos Moure Lopez (ACSO)
• Curs 2023/24

cabo una evaluación del rendimiento de la versión original, se identifican los cuellos de botella y se identifican alternativas potenciales de mejora. Una de estas alternativas, que resulta ventajosa, es el uso de las instrucciones vectoriales o SIMD (*Single Instruction, Multiple Data*) que proporcionan los procesadores modernos. La evaluación experimental de las nuevas propuestas de implementación muestra mejoras de hasta 7 veces respecto a la implementación original.

La estructura del trabajo se divide en varias secciones. Primero, se establecen los objetivos y los criterios que guiarán el desarrollo del proyecto. Luego, se describe la metodología utilizada para alcanzar dichos objetivos. A continuación, se ofrece un breve repaso del estado actual de las técnicas de optimización y de las optimizaciones en algoritmos de alineamiento de secuencias. Después, se explican los conocimientos previos necesarios para comprender este trabajo. Posteriormente, se caracteriza el rendimiento del algoritmo WFA y se presenta una implementación vectorizada de dicho algoritmo, seguida de la evaluación de su desempeño en una plataformas de Computación de Alto Rendimiento (HPC). Finalmente, se discuten las contribuciones de este estudio al campo y se sugieren posibles direcciones para futuras investigaciones.

2 OBJETIVOS

El propósito de este proyecto es acelerar el algoritmo de alineamiento de secuencias *Wavefront* en los procesadores recientes de alto rendimiento. En concreto, se evaluarán las mejoras en Marenostrum5 y en ordenadores del departamento DACSO de la UAB. El objetivo es ofrecer una versión del algoritmo WFA que aproveche eficientemente las capacidades de estos procesadores. Para lograrlo, se establecen los siguientes sub-objetivos específicos:

- Desarrollar y optimizar diferentes versiones del algoritmo WFA.
- Evaluar el rendimiento de estas versiones en los procesadores de Marenostrum5 y UAB.
- Desarrollar una herramienta para visualizar estadísticas de las pruebas de rendimiento, facilitando la comparación de resultados.

3 METODOLOGÍA

Para este proyecto se emplean dos metodologías diferentes. Primero, utilizamos la metodología en cascada para el proyecto en su conjunto y, segundo, la metodología en espiral para cada fase específica del mismo.

En primer lugar, para garantizar el correcto desarrollo del proyecto, optamos por la metodología en cascada debido a la clara dependencia entre los objetivos. Es decir, si uno de ellos no se completa, no podremos avanzar al siguiente. Cada fase del proyecto corresponde a uno de los objetivos que hemos definido previamente, comenzando por familiarizarnos con el algoritmo Wavefront y adquirir todo el conocimiento teórico necesario para llevar a cabo el proyecto.

En segundo lugar, la implementación de diferentes versiones del programa utiliza una metodología en espiral. Partimos de un código base, escalar, y realizamos modificaciones incrementales, de forma iterativa, que vamos validando y evaluando. Las estrategias que mejoran el rendimiento se consolidan, y se prueban nuevas estrategias hasta conseguir la versión vectorial (SIMD) más rápida.

Para identificar los elementos del programa que limitan el rendimiento utilizamos la estrategia que denominamos *diagnóstico diferencial*. Se basa en eliminar o modificar ciertas instrucciones del código para evaluar cuánto afectan al rendimiento, y si representan un cuello de botella significativo. Esto nos permite encontrar nuevas oportunidades de optimización.

Para facilitar la metodología descrita anteriormente, se emplea una interfaz web, diseñada específicamente para este proyecto, que permite visualizar las mejoras de rendimiento entre las diferentes versiones. El flujo de trabajo simplificado de esta interfaz lo podemos ver en la Fig. 1. El primer paso es seleccionar un archivo fuente y compilarlo. Luego se selecciona el conjunto de datos de entrada y se ejecuta el programa. Se utiliza la herramienta *perf* para extraer métricas de rendimiento de la ejecución, como el tiempo total (medido en segundos y en ciclos de reloj), el número de instrucciones ejecutadas, o el número de instrucciones de salto mal predichas. Finalmente, se puede comparar el rendimiento de diferentes ejecuciones, y se pueden generar gráficos automáticamente, como el mostrado en la Fig. 4.

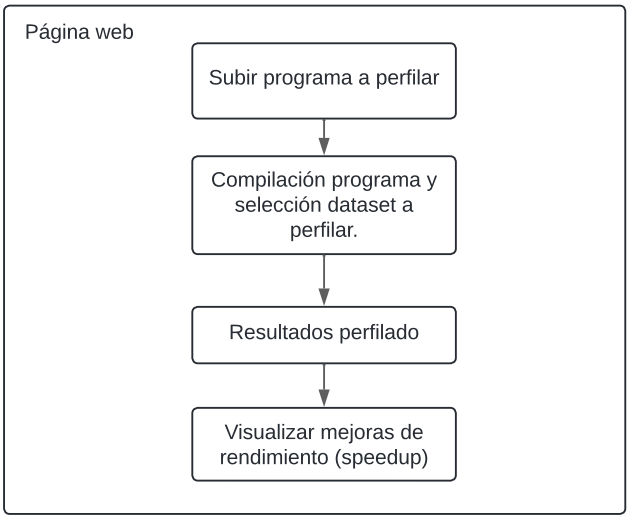

Fig. 1: Flujo de trabajo de la herramienta de visualización de resultados.

4 ESTADO DEL ARTE

Las características del hardware de los procesadores están evolucionando rápidamente. Durante la era de Moore, el desarrollo de software se ha centrado en reducir el tiempo de desarrollo de aplicaciones, en lugar de optimizar su tiempo de ejecución [5]. Esto ha dado lugar a programas a menudo ineficientes, y que no explotan plenamente las capacidades avanzadas de los procesadores, como el paralelismo y las unidades vectoriales. Estos factores han incrementado la relevancia de la ingeniería de rendimiento del software, que busca estructurar el software para mejorar su velocidad de ejecución. Para lograrlo, los ingenieros de rendimiento deben ajustar los programas para aprovechar estas características. Las técnicas de vanguardia incluyen

el aprovechamiento del paralelismo, la optimización de la jerarquía de memoria, entre otras estrategias [6]. Entre las técnicas más destacadas [7] se encuentran las instrucciones vectoriales o SIMD (Single Instruction Multiple Data), optimizaciones de bucles (fusión, desenrollado), *tiling*, *divide-and-conquer*, la eliminación de instrucciones de salto condicional, ...

Existen algunas implementaciones del algoritmo clásico de alineamiento de secuencias que usan instrucciones SIMD [8] para acelerar los cálculos. Por ejemplo, la implementación de Farrar [9] utiliza las instrucciones SSE de Intel. Esta implementación rompe las dependencias, reorganiza los datos y calcula las puntuaciones de manera especulativa. Por otro lado, la implementación de Wozniak [10] reorganiza los datos para exponer el paralelismo en la dimensión antidiagonal de la matriz de puntuación.

5 CONOCIMIENTOS PREVIOS

En esta sección se presenta una breve introducción al problema del alineamiento de secuencias de ADN utilizando la distancia de edición. Luego, se describe el algoritmo de alineamiento de secuencias Wavefront (WFA). Finalmente, se explican las características más relevantes de los procesadores actuales.

5.1. Alineamiento de secuencias biológicas

El alineamiento de secuencias es una operación fundamental en la bioinformática y el análisis de datos biológicos. Debido al auge y desarrollo de las tecnologías de secuenciación, la comparación de secuencias se ha convertido en una operación clave en muchos métodos de análisis genómico. Básicamente, los algoritmos y herramientas de alineamiento de secuencias se utilizan para comparar y encontrar similitudes (o diferencias) entre dos o más secuencias de ADN, ARN o proteínas. Con ello, se busca determinar diferencias entre individuos, mutaciones causantes de enfermedades o similitudes entre cepas de bacterias o virus.

Alinear dos secuencias, $P = p_0, \ldots, p_{n-1}$ (*pattern*) y $T = t_0, \ldots, t_{m-1}$ (*text*), de longitudes n y m respectivamente, consiste en determinar el mínimo número de operaciones de edición (inserción, eliminación y sustitución) necesarias para transformar T en P. Cada operación tiene asociado un coste o peso, que depende de la distancia de alineamiento usada. En el caso de la distancia de edición (también conocida como *Levenshtein distance* [11]) cada operación tiene un coste unitario. El objetivo del algoritmo de alineamiento es computar la secuencia de operaciones de edición (*alineamiento*) de coste global mínimo (*distancia* o *score*) para transformar una secuencia en la otra.

Habitualmente, el problema de alineamiento de secuencias se resuelve usando algoritmos de programación dinámica (*Dynamic Programming* o DP). Esta técnica descompone problemas complejos en subproblemas más simples y luego almacenar los resultados parciales para evitar recalcularlos. Los algoritmos más comunes que se basan en esta técnica son Needleman-Wunsch y Smith-Waterman [2, 3]. Estos algoritmos construyen una matriz M, donde la primera fila y columna representan las secuencias a alinear, y cada celda de la matriz contendrá el menor coste de alineamiento hasta ese punto, para una cierta función de distancia. Cada celda $M_{i,j}$ se calcula a partir de las celdas vecinas ($M_{i-1,j-1}, M_{i,j-1}, M_{i-1,j}$) usando la Eq.1 de manera recurrente. Una vez que se completa la matriz, se reconstruye el alineamiento óptimo rastreando desde la celda final ($M_{n,m}$) hacia la inicial ($M_{0,0}$). Este proceso tiene una complejidad temporal y espacial de $O(n^2)$ (asumiendo que $n \approx m$), es decir, tiene una complejidad cuadrática.

$$M_{i,j} = \min \begin{cases} (p_i \neq t_j) + M_{i-1,j-1} & \text{(Sustitución)} \\ 1 + M_{i,j-1} & \text{(Eliminación)} \\ 1 + M_{i-1,j} & \text{(Inserción)} \end{cases} \tag{1}$$

5.2. Algoritmo Wavefront (WFA)

El algoritmo de alineamiento Wavefront (WFA) [4] es considerado el estado del arte en el problema de alineamiento de secuencias. También es un algoritmo de programación dinámica que, en lugar de calcular todos los valores de la matriz M usada por los algoritmos clásicos, se concentra en las diagonales más relevantes, que contienen los alineamientos más prometedores. De este modo se mejora significativamente la complejidad en tiempo y en espacio.

WFA aprovecha que las matrices de programación dinámica incrementan el valor a lo largo de la diagonal de manera monótona para reducir la complejidad temporal a $O(ns)$, siendo n el tamaño de la secuencia más grande y s el valor del coste mínimo del alineamiento, o *score*. Este enfoque también permite reducir el espacio en memoria a $O(s^2)$. En la práctica, s suele ser mucho menor que n, lo que hace que WFA supere en rendimiento a los algoritmos clásicos.

Originalmente, WFA se formuló para utilizar la distancia *gap-affine*, pero puede ser aplicado a múltiples funciones de distancia. En este trabajo se usará la distancia de edición y se usará la formulación de las ecuaciones para el wavefront definidas en [12] y [13], pero las ideas propuestas en este trabajo son igual de válidas al usar otras métricas de distancia.

El algoritmo WFA representa de forma compacta la información relevante de la matriz M usada por los algoritmos clásicos. Se utiliza una nueva matriz, $\tilde{W}$, que es una serie de frentes de onda $\tilde{W}_e$, de tamaño creciente $(2e + 1)$, que codifican la posición de las celdas con un coste e y que están más cerca de la celda final, $M_{n,m}$. En concreto, el valor $\tilde{W}_{e,k}$ indica la posición de la celda dentro de la diagonal k de la matriz M que tiene un coste e y está más cerca de la celda final.

A partir de esta nueva representación se reformulan las ecuaciones de programación dinámica, que de forma recursiva describen cómo obtener la solución, y se muestran a continuación como las ecuaciones 2 y 3. El término LCP significa *Longest Common Prefix*, y es una función que indica el número de caracteres consecutivos coincidentes entre dos secuencias. En la Eq.2 se aplica la función LCP a dos subsecuencias de las secuencias comparadas, P y T, que comienzan en posiciones codificadas dentro del propio frente de onda $\tilde{W}_e$. Tras obtener la distancia mínima, s, los valores utilizados en el cálculo y representados en $\tilde{W}$ permiten rastrear las operaciones que generan la distancia mínima, y obtener el alineamiento de las secuencias.

$$\tilde{W}_{e,k} = \tilde{W}_{e,k} + LCP(p_{\tilde{W}_{e,k}-k\cdots n}, t_{\tilde{W}_{e,k}\cdots m}) \quad (2)$$

$$\tilde{W}_{e+1,k} = \max \begin{cases} \tilde{W}_{e,k+1} & \text{(Eliminación)} \\ 1 + \tilde{W}_{e,k} & \text{(Sustitución)} \\ 1 + \tilde{W}_{e,k-1} & \text{(Inserción)} \end{cases} \quad (3)$$

El algoritmo 1 es una descripción computacional del proceso completo, en el que las dos operaciones descritas en las ecuaciones anteriores se denominan funciones *extend* y *next*. Comenzando por $\tilde{W}_{0,0} = 0$, es decir, la codificación equivalente a la esquina superior izquierda de la matriz M, el algoritmo calcula progresivamente los frentes de ondas $\tilde{W}_e$, para $e = 0, 1, 2, ..., s$, aplicando repetidamente las funciones *extend* y *next*, hasta alcanzar la codificación equivalente a la esquina inferior derecha de la matriz M.

Algorithm 1 WFA: alineación con distancia de edición

function $align(\tilde{W}_e, \tilde{W}_{e+1}, P, T)$
  $\tilde{W}_{0,0} \leftarrow 0$
  $extend(\tilde{W}_{0,0}, P, T)$
  e $\leftarrow 0$
  while $\tilde{W}_{e,m-n} \neq m$ **do**
    $next(\tilde{W}_e, \tilde{W}_{e+1})$
    e $\leftarrow$ e +1
    $extend(\tilde{W}_e, P, T)$
  end while
end function

El algoritmo 2 define la función *extend*, que se aplica a todos los elementos de un frente de onda, $\tilde{W}_e$. Para cada diagonal k del frente de onda, el bucle interior del programa calcula el LCP indicado en la Eq. 2, entre dos subsecuencias que comienzan, respectivamente, en las posiciones v y h de las secuencias originales, P y T. Este bucle interno compara los caracteres de las dos secuencias de uno en uno, mientras sean iguales. El número total de caracteres iguales consecutivos se usa para actualizar el nuevo valor del frente de onda, $\tilde{W}_{e,k}$. Veremos más adelante que este bucle es el más importante en cuanto al rendimiento de la ejecución del algoritmo.

Algorithm 2 WFA *extend* function

function $extend(\tilde{W}_e, P, T)$
  for k $\leftarrow -e$ **to** e **do**
    v $\leftarrow \tilde{W}_{e,k}$−k
    h $\leftarrow \tilde{W}_{e,k}$
    while $\mathrm{P}_v = \mathrm{T}_h$ **do**
      v $\leftarrow$ v+1
      h $\leftarrow$ h+1
      $\tilde{W}_{e,k} \leftarrow \tilde{W}_{e,k}$+1
    end while
  end for
end function

Una vez se ha extendido diagonalmente la posición de todos los puntos de un frente de onda, si no se ha alcanzado la esquina inferior derecha de la matriz, se utiliza la función *next* para generar la siguiente posición de todos los puntos de un nuevo frente de onda W_{e+1}. El algoritmo 3 muestra una versión ligeramente modificada de la expresión de la Eq. 3, que evita una operación de suma.

Algorithm 3 WFA *next* function

function $next(\tilde{W}_e, \tilde{W}_{e+1})$
  for k $\leftarrow -(e+1)$ **to** $e+1$ **do**
    $\tilde{W}_{e+1,k} \leftarrow \max\{\max\{\tilde{W}_{e,k-1}, \tilde{W}_{e,k}\}+1, \tilde{W}_{e,k+1}\}$
  end for
end function

La figura 2 muestra un ejemplo de alineamiento de las secuencias *GAATA* y *GATTACA*. A la izquierda de la figura se muestra la matriz de programación dinámica usada por los algoritmos clásicos, con algunas celdas numeradas, en las que el número indica el coste mínimo, e, hasta ese punto. La diagonal central de esta matriz se numera como k=0; las diagonales superiores se numeran k=1, 2 ...; y las diagonales inferiores como k= -1, -2 ... A la derecha de la figura se muestra la representación de esta misma información utilizada por el algoritmo WFA, que es una serie de frentes de onda $\tilde{W}_e$, mostrados como columnas verticales, que codifican la posición de las celdas con un coste o distancia de edición e. Por ejemplo, $\tilde{W}_1$ contiene los valores 4, 5 y 2 (de arriba a abajo), que indican, respectivamente, las posiciones dentro de las diagonales k=-1, 0 y 1 con un coste e=1.

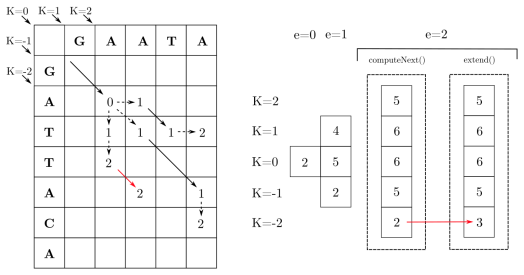

Fig. 2: Ejemplo sencillo que muestra el funcionamiento de WFA (extraído de [13]).

5.3. Rendimiento en un procesador actual

Los procesadores modernos de propósito general son máquinas extremadamente complejas, con numerosas características que afectan al rendimiento de la ejecución de los programas. Entre estas características se destacan la jerarquía de memoria, el uso del paralelismo a nivel de instrucciones y a nivel de datos, y la predicción de saltos. A continuación, se detallan estos aspectos esenciales.

Los procesadores actuales optimizan el **acceso a la memoria** mediante el uso de memorias rápidas y pequeñas, denominadas *memorias caché*, organizadas de forma jerárquica. El procesador, de forma automática, mueve datos entre la memoria principal y las memorias caché para intentar que la mayoría de las peticiones de lectura/escritura que hacen los programas se resuelvan en las memorias caché más rápidas, y así reducir el tiempo total de acceso en comparación al tiempo de la memoria principal. El algoritmo automático que gestiona el movimiento de los datos en la jerarquía de memoria basa su efectividad en las propiedades de localidad de acceso a memoria que experimentan la mayoría de los programas: localidad temporal, que implica que si un dato

ha sido accedido recientemente, es probable que sea accedido nuevamente en un futuro cercano; y localidad espacial, que se refiere a la tendencia de los programas a acceder a direcciones de memoria cercanas entre sí.

El paralelismo a nivel de instrucciones (ILP, del inglés *Instruction-Level Parallelism*) se refiere a la capacidad de un procesador para ejecutar múltiples instrucciones simultáneamente. Los procesadores modernos implementan varias técnicas para aprovechar este paralelismo y así mejorar el rendimiento en la ejecución de programas. Una de las técnicas fundamentales para aprovechar el ILP es el *pipelining*. Esta técnica permite al procesador solapar parcialmente la ejecución concurrente de varias instrucciones. Otra técnica esencial es la ejecución fuera de orden. En lugar de ejecutar las instrucciones en el orden especificado por el programa, el procesador reordena su ejecución para así aumentar el uso de sus recursos de cómputo, siempre y cuando se mantengan las dependencias de datos. Esto significa que mientras una instrucción espera los resultados de una instrucción anterior, el procesador puede ejecutar otras instrucciones independientes. La reordenación de las instrucciones se realiza de forma dinámica, en tiempo de ejecución, sin necesitar ayuda del compilador.

La ejecución especulativa es otra técnica avanzada utilizada para aprovechar mejor el ILP. El procesador ejecuta instrucciones antes de estar completamente seguro de que su ejecución será necesaria. Esto ocurre cuando se ejecutan estructuras de control, como bloques *if-else*, de modo que el procesador predice el resultado de una condición y empieza a ejecutar las instrucciones correspondientes antes de que la condición sea evaluada. Concretamente, los procesadores actuales implementan técnicas de predicción aplicadas a las instrucciones de salto, que son las instrucciones que alteran el flujo de ejecución del programa en función del resultado de una condición. La precisión de la predicción de saltos es crucial, ya que una predicción incorrecta supone descartar las instrucciones que se han ejecutado de forma especulativa en un camino mal predicho, y por tanto supone reducir el rendimiento de la ejecución.

Las técnicas descritas anteriormente, e implementadas a nivel interno en los procesadores, son transparentes respecto a la funcionalidad de los programas, el diseño del algoritmo y de las estructuras de datos, pero tienen un impacto crucial en el rendimiento. Para lograr que sean efectivas, el algoritmo debe proporcionar el paralelismo potencial a nivel de instrucciones que permita al procesador ejecutar el máximo número de instrucciones de forma simultánea. Además, se deben evitar estructuras de control condicionales difíciles de predecir. Finalmente, tiene que utilizar estructuras de datos a cuyos elementos se acceda en un orden que maximice los dos tipos de localidad que se han descrito.

Los procesadores actuales también aprovechan el paralelismo a nivel de datos que ofrecen muchos programas. Se trata de realizar una misma operación simultáneamente sobre múltiples datos, y obtener múltiples resultados. Este tipo de procesamiento agrupa los datos en pequeños vectores (registro vectorial), y realiza operaciones directamente sobre estos vectores. Los procesadores actuales ofrecen soporte para este tipo de paralelismo a través de instrucciones especiales conocidas como instrucciones vectoriales o SIMD (*Single Instruction, Multiple Data*). Por ejemplo, para sumar una larga lista de números, en lugar de sumarlos de uno en uno, se pueden usar instrucciones SIMD para sumar varios números con una sola instrucción, tal como se ilustra en la Fig. 3. La cantidad de datos que pueden ser procesados simultáneamente está determinada por el tamaño del registro vectorial y el tamaño de cada dato.

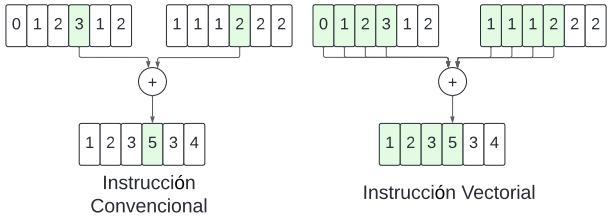

Fig. 3: Suma de vectores usando instrucciones convencionales e instrucciones vectoriales (SIMD) de 4 carriles.

Las instrucciones SIMD son especialmente útiles en aplicaciones que requieren cálculos intensivos, como el procesamiento de imágenes, el procesamiento de señales digitales o la simulación física. En estas aplicaciones, es común realizar operaciones idénticas en grandes conjuntos de datos de manera simultánea, lo que conduce a mejoras significativas en el rendimiento y la eficiencia del procesador.

Tanto AMD como Intel implementan instrucciones SIMD en sus arquitecturas y procesadores. A lo largo del tiempo, han expandido la variedad de tipos soportados, aumentando el tamaño de los registros vectoriales e incluso introduciendo operaciones que anteriormente estaban exclusivamente reservadas para computadoras de alto rendimiento (como los accesos indexados a memoria, *gather*). El nombre técnico de estas extensiones al repertorio de instrucciones, en orden cronológico, es SSE, AVX, AVX2 y AVX-512. La principal diferencia entre AVX2 y AVX512 radica en que los registros pasan de 256 bits a 512 bits.

6 ANÁLISIS Y CARACTERIZACIÓN DEL RENDIMIENTO DE WFA

Los algoritmos clásicos de alineamiento de secuencias tienen un procesamiento regular, y utilizan una cantidad constante de instrucciones para calcular el contenido de cada celda de la matriz de programación dinámica, M. Además, las instrucciones de salto asociadas en los bucles de tipo *for* de estos algoritmos tienen un número de iteraciones predecible y accesos a memoria en posiciones consecutivas. Todo esto permite diseñar implementaciones de programas que aprovechan eficientemente las características de los procesadores modernos, como el paralelismo a nivel de instrucciones y de datos (instrucciones SIMD), la predicción de saltos, y la localidad espacial de acceso a memoria. En consecuencia, el rendimiento de la ejecución depende únicamente del tamaño de las secuencias, $O(nm)$, y estará limitado por la capacidad de cómputo del procesador en el caso de que la matriz M no sea muy grande, o, en caso contrario, estará limitado por la cantidad de datos que puede proporcionar el sistema de memoria por unidad de tiempo, es decir, por su ancho de banda.

En contraste, el número de operaciones realizado por WFA no solamente depende del tamaño de las secuencias, sino que también depende de su similitud. Secuencias de

igual longitud pueden dar lugar a volúmenes de trabajo computacional muy diferentes. Por un lado, el número de celdas calculadas en la nueva representación de la matriz de frentes de onda, $\tilde{W}$, es proporcional a $O(ns)$, el tamaño de la secuencia más grande y el coste mínimo de alineamiento (*score*), que no se conoce a priori. Por otro lado, la operación LCP (*Longest Common Prefix*) realiza un trabajo proporcional al número de caracteres consecutivos iguales en dos segmentos de las secuencias, que también es variable y difícil de predecir.

El algoritmo WFA ejecuta repetidamente las funciones *next* y *extend*. La función *next* realiza un trabajo regular, con mucho paralelismo potencial, tanto a nivel de instrucciones como a nivel de datos, saltos condicionales predecibles, y acceso a posiciones consecutivas en la memoria. En cambio, la función *extend* presenta características irregulares que dificultan implementaciones eficientes en los procesadores actuales. Por un lado, el cálculo del LCP entre dos secuencias contiene un bucle de tipo *while* que realiza un número de iteraciones difícilmente predecible, ya que depende de los datos de entrada. Por otro lado, los accesos a los datos de las secuencias no son siempre contiguos. Todo esto dificulta el uso del paralelismo, tanto a nivel de instrucciones como a nivel de datos, y el uso de la memoria.

A pesar de la irregularidad computacional de la función *extend*, hemos identificado técnicas para mejorar su rendimiento. Hemos instrumentado el código de WFA para analizar la irregularidad de la función *extend* para varios conjuntos de datos de secuencias reales, representativos de las tecnologías de secuenciación actuales, cuyas características se explican en detalle en la sección 8.2. Se han medido los valores de la operación LCP en la función *extend* y se ha generado un histograma con la distribución de estos valores, que se muestra en la Fig. 4 para tres conjuntos de datos, etiquetados como *Nanopore*, *PacBio* y *Illumina*.

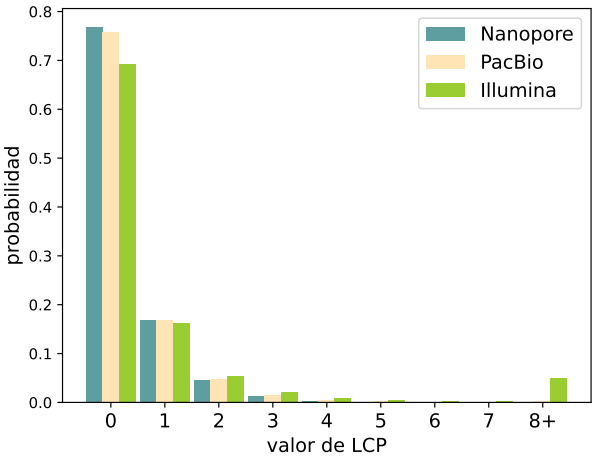

Fig. 4: Distribución de probabilidad de los valores obtenidos con la operación LCP

El caso LCP $= 0$, cuando el primer carácter de cada secuencia comparada es diferente, ocurre entre el 69 % y el 77 % de las veces, y es el caso que se debería priorizar al optimizar el programa. El caso LCP $= 4$ o más, tener al menos los 4 primeros caracteres iguales entre las secuencias, representa un 0.5-1.2 % para *PacBio* y *Nanopore*, mientras que para *Illumina* representa un 7 %. El caso LCP $= 8$ o más, supone menos de un 0.3 % para *PacBio*, menos de 0.04 % para *Nanopore* y alrededor del 6 % para Illumina. En breve explicaremos cómo utilizar estos resultados para mejorar el rendimiento, y por qué al utilizar datos de tipo *Illumina* obtenemos mejoras de rendimiento más pequeñas que con *PacBio* o *Nanopore*.

7 Optimización de WFA

Esta sección describe las optimizaciones realizadas a la función *extend* de WFA para aprovechar mejor las características de los procesadores actuales. Brevemente, se utiliza el paralelismo de datos tanto al comparar caracteres, como al procesar elementos consecutivos de un frente de onda. En el primer caso se utilizan instrucciones escalares para procesar números enteros de 4 u 8 bytes y en el segundo caso se utilizan instrucciones SIMD.

7.1. Procesar caracteres en paralelo

Nuestra propuesta consiste en comparar directamente bloques de 4 u 8 caracteres por secuencia, almacenados en una misma palabra, y utilizar una combinación de varias instrucciones genéricas para encontrar el primer carácter diferente entre los bloques. Como cada carácter se codifica como un byte, denominamos a esta propuesta `byte-parallel`. El Algoritmo 4 muestra el caso en que se usan bloques de 4 caracteres. Sigue siendo necesario un bucle interno (*while*), que itera leyendo bloques de 4 caracteres consecutivos de las secuencias P y T, mientras todos los caracteres de los dos bloques sean iguales. La comparación de 4 caracteres a la vez se realiza con la operación XOR ($\oplus$). Una vez que aparece alguna diferencia, se necesita encontrar la posición del primer carácter diferente. Para ello se utiliza una operación que encuentra la posición del primer bit con valor 1 en el resultado de la operación $\oplus$, y que se suele denominar *trailing zeros count* (`tzcnt`). Como cada carácter se codifica con 8 bits (ASCII), la posición obtenida se ha de dividir por 8.

Algorithm 4 WFA $extend_4$ function

function $extend_4(\tilde{W}_e, P, T)$
  for k $\leftarrow -e$ **to** e **do**
    v $\leftarrow \tilde{W}_{e,k}$ − k
    h $\leftarrow \tilde{W}_{e,k}$
    cmp $\leftarrow \mathrm{P}_{v\cdots v+3} \oplus \mathrm{T}_{h\cdots h+3}$
    while cmp $= 0$ **do**
      $\tilde{W}_{e,k} \leftarrow \tilde{W}_{e,k}$+4
      v $\leftarrow$ v+4
      h $\leftarrow$ h+4
      cmp $\leftarrow \mathrm{P}_{v\cdots v+3} \oplus \mathrm{T}_{h\cdots h+3}$
    end while
    lcp $\leftarrow$ tzcnt(cmp)/8
    $\tilde{W}_{e,k} \leftarrow \tilde{W}_{e,k}$+lcp
  end for
end function

El objetivo de mejora de rendimiento que se busca con esta nueva propuesta es que el comportamiento del bucle interior sea mucho más regular, y que se mejore significativamente la efectividad del predictor de saltos. El análisis mostrado al final del apartado anterior indicaba que los

casos en que LCP≥4 (LCP≥8) son muy poco frecuentes, así que el salto se producirá muy pocas veces. Al realizar 4 (8) comparaciones de una sola vez, y al leer de memoria 4 (8) Bytes de una sola vez, se reduce el número total de instrucciones ejecutadas por el programa cuando el valor obtenido de LCP es relativamente grande. Sin embargo, como se han añadido instrucciones adicionales para encontrar el primer carácter diferente (`tzcnt` y división por 8) en los casos más frecuentes (LCP $= 0$ o 1) el número total de instrucciones ejecutadas aumentará. Aunque pueda parecer sorprendente, en un procesador actual no siempre se consigue ir mas rápido al ejecutar menos instrucciones.

7.2. Vectorización del código (SIMD)

En esta sección mostramos cómo acelerar la ejecución de las funciones *next* y *extend* usando paralelismo de tipo vectorial (SIMD), y cómo es conveniente fusionar ambas funciones para lograr la máxima mejora.

7.2.1. Vectorizacion de la función *next*

El algoritmo 3 muestra que la función *next* genera cada nuevo elemento del frente de onda W_{e+1} a partir de tres valores consecutivos del frente de onda previo, W_e, realizando dos operaciones de máximo y una suma. Como cada nuevo elemento se puede calcular de manera independiente al resto de los nuevos elementos, es decir, de forma simultánea, la vectorización del bucle es muy directa, tal como se visualiza en el esquema de la Fig. 5. Los círculos del gráfico representan operaciones SIMD binarias con vectores de 4 datos (o con 4 carriles). La simplicidad del patrón de cómputo, que habitualmente se denomina de tipo *stencil*, permite que el compilador sea capaz de vectorizar el código de forma automática.

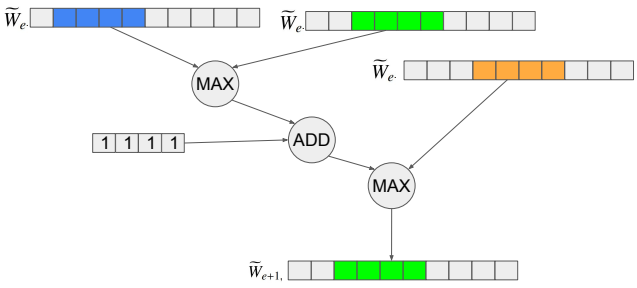

Fig. 5: Operación *next* con vectores de 4 datos (carriles)

7.2.2. Vectorización de la función *extend*

Tal como se muestra en la Fig. 6, la vectorización de la función *extend* es bastante más compleja que la de la anterior función, y esto explica que el compilador no sea capaz de automatizarla. Por tanto, es necesario utilizar funciones especiales, denominadas intrínsecas (*intrinsics*), que se proporcionan en lenguaje C/C++ y se convierten de forma directa en instrucciones SIMD. Por simplicidad, vamos a describir la operación vectorial *extend* asumiendo que se usan 4 carriles, tal como se muestra en la Figura.

El primer paso del algoritmo requiere generar los 4 índices a cada una de las dos secuencias, P y T, y leer simultáneamente 4 bloques diferentes de cada secuencia. La

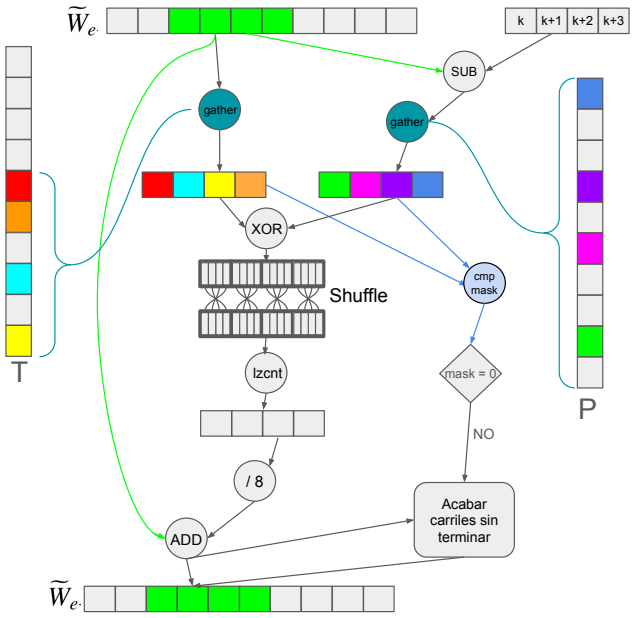

Fig. 6: Operación *extend* con vectores de 4 datos (carriles)

instrucción vectorial denominada *gather* es la que realiza las 4 lecturas a memoria, y su particularidad es que las 4 direcciones a memoria se especifican usando 4 índices diferentes (almacenados en un vector, también de 4 carriles). Esto permite que las 4 direcciones a memoria no sean necesariamente contiguas, es decir, que sean dispersas. Aunque se trata de una instrucción muy poderosa, mostraremos más adelante que el rendimiento que ofrecen los procesadores al ejecutar esta instrucción no es muy alto. Aunque existe una versión de la instrucción *gather* en la que cada carril lee un bloque de 8 caracteres consecutivos, la versión que lee bloques de 4 caracteres consecutivos usa el doble de carriles, y es la que proporciona mejor rendimiento. Dicho en otras palabras, aumentar el paralelismo a nivel de bytes de 4 a 8 mejora menos el rendimiento que aumentar el número de carriles de las instrucciones vectoriales.

Otro problema a resolver es que ocasionalmente algunos carriles de la operación vectorial necesitan iteraciones adicionales, porque el valor del LCP para ese carril es mayor al número de caracteres comparados en un solo paso de la operación. Nuestra propuesta comprueba con una condición sencilla si todos los carriles se han completado en un único paso, que es el caso más probable, y el que conviene optimizar. En caso contrario, se diseña un bucle que identifica uno a uno qué carriles hay que extender y que aplica la versión escalar `byte-parallel` a esos carriles.

La instrucción vectorial para la operación *tzcnt* (*trailing zeros count*) no existe en los procesadores actuales, y su funcionalidad se debe emular. En procesadores que disponen de una instrucción SIMD muy similar (AVX512), denominada *lzcnt* (*leading zeros count*), se utiliza una instrucción *shuffle* previa para cambiar el orden de los valores (*endianess*). En procesadores que no disponen de esa instrucción (AVX2), es necesario usar una combinación de varias instrucciones que acabe realizando la misma función.

7.2.3. Fusionar *next* y *extend*

El análisis de la ejecución de la función *extend* vectorizada indica que la capacidad del procesador para ejecutar instrucciones *gather* limita el rendimiento. En otras palabras, el cuello de botella al rendimiento son la operaciones *gather*, mientras que los recursos del procesador para ejecutar otros tipos de instrucciones de cómputo están infrautilizados. Al no existir alternativas más rápidas a la instrucción *gather*, hemos considerado fusionar las funciones *next* y *extend*, ver algoritmo 5, y así lograr dos beneficios. En primer lugar, se reduce el número de lecturas y escrituras a memoria necesarias para acceder a los frentes de onda, $\tilde{W}_e$. En segundo lugar, se puede realizar el cómputo asociado a la función *next* en paralelo a las operaciones *gather* de la función *extend*, y así aumentar la utilización de los recursos del procesador.

Algorithm 5 WFA *next-extend* function

function $next{-}extend(\tilde{W}_e, \tilde{W}_{e+1})$
  for k $\leftarrow -(e+1)$ **to** $e+1$ **do**
    offset $\leftarrow \max\{\max\{\tilde{W}_{e,k-1}, \tilde{W}_{e,k}\}+1, \tilde{W}_{e,k+1}\}$
    v, h $\leftarrow$ offset - k, offset
    while $\mathrm{P}_v = \mathrm{T}_h$ **do**
      v, h, offset $\leftarrow$ v+1, h+1, offset+1
    end while
    $\tilde{W}_{e+1,k} \leftarrow$ offset
  end for
end function

El problema de fusionar las dos funciones es que se debe vectorizar de forma explícita la función *next*, mientras que con funciones separadas esta parte era vectorizada de forma automática por el compilador.

8 Evaluación del rendimiento

En esta sección se describen los procesadores y los conjuntos de datos de entrada utilizados, y se presentan, comparan y discuten los resultados de rendimiento de las versiones desarrolladas en este proyecto.

8.1. Entorno de ejecución

Se han realizado ejecuciones en dos tipos de procesadores para identificar posibles diferencias en el efecto de las optimizaciones propuestas. En primer lugar se utilizó el procesador de sobremesa Intel i5-12400 disponible en el Dept. DACSO de la UAB, y de arquitectura relativamente reciente (2022). Adicionalmente, se ha usado el procesador servidor Intel Xeon Platinum 8480+ (2023), que forma parte del supercomputador MareNostrum 5 en el BSC.

Las principales diferencias entre ambos procesadores son el conjunto de instrucciones, la frecuencia de reloj, el tamaño de las memorias caché y el número total de núcleos de cómputo. El procesador i5-12400 opera a una frecuencia más alta (4,4 GHz) que el procesador Xeon (3,0 GHz), pero solamente proporciona el conjunto de instrucciones AVX2, que permite usar 8 carriles de 32 bits por carril (total de 256 bits), mientras que Xeon proporciona también la extensión AVX512, que permite usar 16 carriles. Por otro lado, el procesador Xeon dispone de una memoria caché de último nivel bastante más grande (105MB) que la del procesador i5 (18MB) y contiene un número muy superior de núcleos de cómputo (56 frente a 6). Mientras que los dos primeros factores son relevantes para explicar las diferencias de rendimiento, los dos últimos factores no tienen influencia, ya que los datos caben en el nivel de caché L2 y todas las ejecuciones usan un único hilo (*thread*), y un solo núcleo.

8.2. Conjunto de datos de evaluación

Se han seleccionado conjuntos de datos de entrada reales, obtenidos con varias tecnologías actuales de secuenciación, que son **Nanopore**, **Illumina** y **PacBio**. Las secuencias de estos conjuntos tienen diferentes tamaños y tasas de error, que permitirán evaluar el efecto de estas variaciones en el rendimiento de cada versión del programa. También se han generado secuencias de contenido aleatorio, en las que se han provocado errores de forma artificial, que etiquetamos como el conjunto de datos **Simulados**.

Las secuencias PacBio tienen longitudes que varían entre 201 y 24,640 pares de bases (bps), y alta precisión (tasas de error $\leq 1\,\%$). Las secuencias Nanopore tienen longitudes similares a las PacBio (oscilan entre 104 y 21,708 bps), pero con tasas de error más altas ($\sim 10\,\%$). Las secuencias Illumina son bastante cortas, con longitudes que varían entre 140 y 276 bps, y tienen baja tasa de error ($\sim 1\,\%$). Finalmente, las secuencias simuladas tienen longitudes promedio de 100,000 bps, y un error alrededor del $10\,\%$.

En cada ejecución se alinea un conjunto de secuencias suficientemente grande para que el tiempo total de ejecución sea bastante superior al tiempo de lectura de los archivos y de inicialización de las estructuras de datos.

8.3. Rendimiento i5-12400

La Fig. 7 muestra la mejora de rendimiento, o *speedup*, utilizando el procesador i5-12400 para cada una de las versiones descritas en la sección anterior. La estrategia `byte-parallel` logra mejoras de entre 1.7x y 3.9x respecto a la versión original. Si se añade el uso de instrucciones vectoriales se alcanzan mejoras totales de entre 1.8x y 7.0x. Por tanto, añadir el uso de instrucciones SIMD incrementa el rendimiento entre 1.04x y 1.8x. La fusión de los bucles proporciona una mejora de entre el $2\,\%$ y el $4\,\%$.

Las mejoras de rendimiento varían mucho según el tipo de secuencias de entrada: cuanto más largas son (Nanopore y PacBio) y mayores diferencias hay entre ellas (Nanopore) mayor es la mejora obtenida al usar las estrategias propuestas. En concreto, si las secuencias son muy cortas y el número total de diferencias es muy pequeño, como Illumina, no se aprovecha el paralelismo ni al comparar caracteres ni al procesar de forma vectorial múltiples elementos del frente de onda, y la mejora de rendimiento es relativamente pequeña. Por otro lado, las secuencias de Nanopore, con longitudes similares a las de PacBio, presentan una mayor tasa de error y generan frentes de onda más largos, que se benefician más de la ejecución SIMD.

8.3.1. Procesar caracteres en paralelo

Al procesar 4 u 8 caracteres en paralelo se mejora el rendimiento porque el procesador acierta más veces al predecir

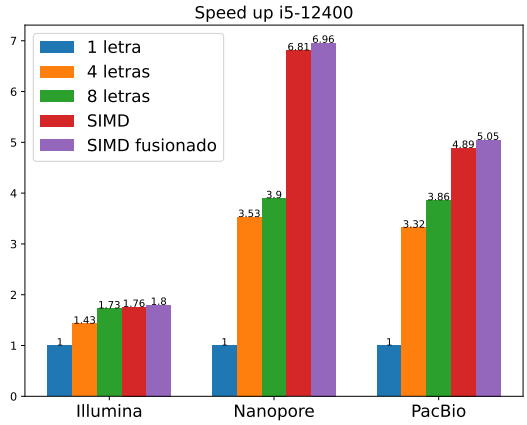

Fig. 7: Mejora de rendimiento en el procesador i5-12400 para las versiones propuestas y tres conjuntos de datos de entrada.

el camino que seguirá la ejecución del programa. La cantidad de errores de predicción se reduce de unos 20 por cada mil instrucciones ejecutadas a valores de entre 1 y 0,2. Los fallos en la predicción por parte del procesador son muy costosos, porque descartan trabajo ya realizado, limitando el promedio de instrucciones ejecutadas por unidad de tiempo. Tal como mostramos en la Fig. 4, al procesar de 4 en 4 o de 8 en 8, la probabilidad de que se encuentre una diferencia en los primeros caracteres comparados es muy alta (en muchos casos superior al 99 %), y el bucle interno no se suele realizar. De este modo, la predicción que hace el procesador es mucho más precisa.

Para implementar esta versión del programa ha sido necesario utilizar instrucciones adicionales y, aunque no se muestra en la figura anterior, en general, el número total de instrucciones ejecutadas ha crecido ligeramente. Sin embargo, estas instrucciones se ejecutan ahora a ritmos de entre 4,6 y 6,5 instrucciones por ciclo de reloj, que prácticamente alcanzan el 80-100 % de la máxima capacidad de ejecución del procesador. Esto indica que el único camino viable para seguir mejorando el rendimiento de la ejecución es reducir el número total de instrucciones ejecutadas.

Comparar de 8 en 8 no supone una mejora importante respecto a comparar de 4 en 4, porque la mayoría de las veces que se comparan los primeros 4 caracteres no es necesario seguir realizando comparaciones. Esto era algo que se podía deducir de los resultados mostrados en la Fig. 4, y que hemos comprobado experimentalmente: el número total de instrucciones ejecutadas al comparar de 8 en 8 apenas decrece un 1 % respecto a comparar de 4 en 4.

8.3.2. Procesamiento SIMD

Hemos argumentado que el cuello de botella del rendimiento era la cantidad total de instrucciones ejecutadas, así que ejecutar instrucciones vectoriales permite reducir esta cantidad de instrucciones en proporción al número de carriles (elementos) del vector SIMD. En el caso del i5-12400, se procesan 8 elementos del *wavefront* simultáneamente. Sin embargo, no podemos esperar una reducción de 8 veces porque se han tenido que añadir instrucciones adicionales para emular el funcionamiento de *lzcnt*, como *shuffle*, e instrucciones para detectar si para alguno de los carriles del vector se necesita seguir realizando comparaciones de caracteres. En este último caso, se ejecutan instrucciones adicionales para identificar los carriles que necesitan procesamiento adicional.

Empíricamente se ha medido que el total de instrucciones ejecutadas se ha reducido entre 4 y 5,3 veces. Sin embargo, las instrucciones se ejecutan a un ritmo más bajo: entre 1,6 y 2,2 instrucciones por ciclo de reloj. Esto explica por qué la mejora total de rendimiento no es superior de 2 veces respecto de la versión `byte-parallel` de 4 caracteres. Usando técnicas de ingeniería de rendimiento, hemos encontrado que las instrucciones *gather*, que utilizan 8 índices para realizar 8 lecturas a posiciones dispersas de la memoria, suponen el cuello de botella en el rendimiento. El problema no es el tiempo, o latencia, que toma la ejecución de la instrucción, sino la falta de capacidad del procesador para solapar la ejecución de muchas instrucciones *gather* (aprovechar correctamente ILP). Mientras que cada nuevo ciclo de reloj el procesador es capaz de iniciar la ejecución de 2 o incluso 3 instrucciones SIMD cualquiera, esto no es cierto para las instrucciones *gather*, y se necesita esperar varios ciclos entre la ejecución de dos instrucciones *gather* independientes.

8.4. Rendimiento en MareNostrum 5

En la figura 8 se muestra la mejora de rendimiento, o *speedup*, en los procesadores Xeon 8480+, usando los dos repertorios SIMD disponibles, AVX2 y AV512. Solo se presentan resultados de las versiones fusionadas, que siempre son más rápidas. Hemos sustituido las secuencias de Illumina (demasiado cortas) por un conjunto de secuencias simuladas de gran longitud (100.000 bps), para así evaluar el rendimiento en el caso en que los datos no caben en el primer nivel de caché del procesador.

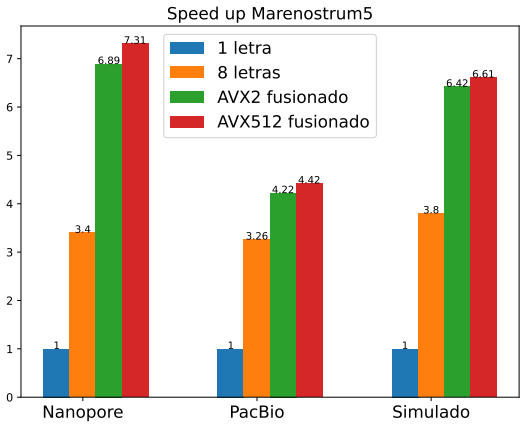

Fig. 8: Mejora de rendimiento en MareNostrum 5 para versiones seleccionadas del programa y tres conjuntos de datos de diferentes características.

Los tiempos totales de la ejecución son algo inferiores a los obtenidos con el procesador i5-12400, porque el procesador X8480+ funciona a una menor frecuencia de reloj, pero las mejoras de rendimiento son muy similares: (1) entre 3 y 4 veces gracias al uso de paralelismo al comparar

caracteres; y (2) entre 1,4 y 2,2 veces al añadir el uso de instrucciones vectoriales. Este procesador dispone de un repertorio más amplio de instrucciones SIMD que permiten reducir más el total de instrucciones ejecutadas, entre 4,4 y 6 veces al usar AVX2 y entre 6,6 y 12,7 veces al usar AVX512. Sin embargo, la mejora de rendimiento por usar instrucciones SIMD sigue siendo muy baja. Una vez más, la capacidad de ejecución para las instrucciones *gather* limita el rendimiento del programa (es el cuello de botella).

Las instrucciones AVX512 proporcionan 16 carriles, pero la mejora obtenida respecto a la instrucciones AVX2, de 8 carriles, es 1.06x para Nanopore, 1.05x para PacBio y 1.03x para secuencias simuladas (ultra largas). En este último caso se ha apreciado una pequeña penalización por el hecho de que con cierta frecuencia el procesador necesita acceder a los datos en el segundo nivel de caché.

9 Conclusiones

En este trabajo se han identificado y propuesto modificaciones para acelerar el algoritmo de alineamiento Wavefront (WFA) y se ha evaluado su rendimiento en plataformas de computación de alto rendimiento (HPC). Se ha identificado la función *extend* como la más crítica en cuanto al rendimiento, y el análisis computacional ha permitido identificar oportunidades para aprovechar mejor las características de los procesadores actuales.

Se han propuesto tres estrategias: (1) comparar varios caracteres de las secuencias en paralelo, basándonos en la frecuencia de los valores de la operación **LCP**; (2) usar paralelismo de datos en forma de instrucciones vectoriales (SIMD), y así procesar varios datos de forma simultánea; y (3) fusionar las funciones *next* y *extend* en un único bucle, para reducir el número de instrucciones ejecutadas y aprovechar mejor la capacidad de ejecución paralela del procesador. Las propuestas se han realizado y evaluado usando dos repertorios de instrucciones SIMD, de 256 bits (AVX2) y de 512 bits (AVX512).

La primera estrategia ha conseguido acelerar el algoritmo respecto a la versión original hasta 3.9x en un procesador i5-12400 (UAB) y hasta 3.8x en un procesador Xeon 8480+ (MareNostrum 5). La versión vectorizada mostró mejoras variables según los datos de entrada, el conjunto de instrucciones utilizadas, y el uso de la fusión de bucles, alcanzando hasta 6.96x con AVX2 (i5-12400) y hasta 7.31x con AVX512 (Xeon 8480+).

Es necesario que las secuencias comparadas sean relativamente largas (miles de caracteres) para que las mejoras sean más efectivas. Por ejemplo, para secuencias de 250 caracteres con poco error (Illumina) las mejoras fueron de 1.8x. La eficiencia de la vectorización ha sido relativamente baja, ya que a pesar de reducir la cantidad de instrucciones ejecutadas entre 4,4 y 12,7 veces, la mejora de tiempo apenas ha alcanzado 2x. Hemos identificado las limitaciones de los procesadores para ejecutar la instrucción *gather* como el cuello de botella del rendimiento.

Una línea futura a este trabajo es analizar la efectividad de las propuestas en procesadores de arquitectura RISC-V y ARM, que representan soluciones en las que se suele priorizar la eficiencia energética. Otro camino de investigación es explorar el diseño de mecanismos hardware que proporcionen más rendimiento al ejecutar instrucciones *gather*.

Referencias

[1] M. Alser, J. Lindegger, C. Firtina, N. Almadhoun, H. Mao, G. Singh, J. Gomez-Luna, and O. Mutlu, "From molecules to genomic variations: Accelerating genome analysis via intelligent algorithms and architectures," *Computational and Structural Biotechnology Journal*, vol. 20, pp. 4579–4599, 2022.

[2] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970.

[3] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of Molecular Biology*, vol. 162, no. 3, pp. 705–708, 1982.

[4] S. Marco-Sola, J. C. Moure, M. Moreto, and A. Espinosa, "Fast gap-affine pairwise alignment using the wavefront algorithm," *Bioinformatics*, vol. 37, pp. 456–463, 02 2021.

[5] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl, "There's plenty of room at the top: What will drive computer performance after moore's law?," *Science*, vol. 368, no. 6495, p. eaam9744, 2020.

[6] D. Bakhvalov, *Performance analysis and tuning on modern cpus*. 2020.

[7] Intel, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel, April 2024.

[8] D. T. Popovici, M. G. Awan, G. Guidi, R. Egan, S. Hofmeyr, L. Oliker, and K. Yelick, "Designing efficient simd kernels for high performance sequence alignment," in *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 167–176, 2023.

[9] M. Farrar, "Striped Smith–Waterman speeds database searches six times over other SIMD implementations," *Bioinformatics*, vol. 23, pp. 156–161, 11 2006.

[10] A. Wozniak, "Using video-oriented instructions to speed up sequence comparison," *Bioinformatics*, vol. 13, pp. 145–150, 04 1997.

[11] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals.," *Soviet Physics Doklady*, vol. 10, pp. 707–710, feb 1966. Doklady Akademii Nauk SSSR, V163 No4 845-848 1965.

[12] Q. Aguado Puig, *Accelerating pairwise sequence alignment on GPUs using the Wavefront Algorithm*. PhD thesis, UPC, Facultat d'Informàtica de Barcelona, Departament d'Arquitectura de Computadors, Oct 2022.

[13] Q. Aguado-Puig, S. Marco-Sola, J. C. Moure, D. Castells-Rufas, L. Alvarez, A. Espinosa, and M. Moreto, "Accelerating edit-distance sequence alignment on gpu using the wavefront algorithm," *IEEE Access*, vol. 10, pp. 63782–63796, 2022.