DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Final

Alineamiento de secuencias de ADN

Organización del Computador II

Integrante	LU	Correo electrónico
Octavio Gianatiempo	280/10	ogianatiempo@gmail.com
Tomas Tropea	115/18	tomastropeaa@gmail.com
Bruno Gomez	428/18	brunolml99@outlook.es

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice

1. Introducción

1.1. Biología celular y molecular

La superficie de nuestro planeta está poblada por seres vivos, conjuntos intrincados y complejos de reacciones químicas organizadas que toman materia y energía del medio ambiente para generar copias de ellos mismos. A pesar de su gran diversidad, ¿Qué puede ser más distinto que un tigre, un alga, una bacteria y un árbol?, nuestros ancestros lograron englobar a estas entidades bajo el concepto de "vida" y se maravillaron con ellas aún sin poder explicar lo que el hecho de pertenecer a dicha categoría conlleva y las diferencias fundamentales que implica respecto de los objetos inanimados. Los avances del último siglo en biología han removido esta capa de misterio acerca del funcionamiento de los seres vivos, sin por ello disminuir la curiosidad del hombre respecto de los mismos. Hoy sabemos que todos los organismos vivos están compuestos por células: pequeñas unidades delimitadas por una membrana de lípidos y rellenas con una solución acuosa de sustancias químicas y moléculas que le confieren la habilidad extraordinaria de crear copias de si misma (Alberts y col. 2017).

Dado que las células son las unidades fundamentales de la vida debemos recurrir a la biología celular, que comprende el estudio de la estructura, la función y el comportamiento de las mismas, para buscar respuestas a las preguntas de qué es la vida y cómo funciona. Un mayor entendimiento de las células y su evolución nos permitirá empezar a afrontar las grandes incógnitas de la vida en la tierra como lo son su misterioso origen, su fascinante diversidad y su capacidad para colonizar casi todos los hábitat concebibles. La mayoría de los organismos están compuestos por una única célula. Otros, como los seres humanos, son bastas ciudades multicelulares en las cuales distintos grupos de células realizan funciones especializadas y se comunican mediante intrincados sistemas a distintas escalas. Pero incluso para el agregado de más de 10^{13} células que forman un cuerpo humano, la totalidad del organismo fue generada por divisiones a partir de una única célula. Por lo tanto, esta única célula es el vehículo para toda la información hereditaria que define cada especie (Alberts y col. 2017).

Una gran diversidad de células existe en la naturaleza, pero todas comparten ciertas características. En particular, tienen un ciclo de vida. Es decir que nacen, crecen, se reproducen y mueren. Durante dicho ciclo, las células tienen que tomar muchas decisiones importantes. Por ejemplo, si una célula intenta replicarse antes de haber recolectado los nutrientes necesarios sería desastroso. Sin embargo, las células no tienen cerebros. Las decisiones se manifiestan como resultado de complejas interacciones en redes de reacciones químicas que sintetizan nuevos compuestos, rompen otros para obtener sus componentes o incluso envían señales de que ya es hora de alimentarse o morir. Los algoritmos que controlan la vida de las células son sorprendentemente confiables y complejos, y su comprensión completa todavía se nos escapa. Uno puede pensar la célula como un sistema mecánico intrincado con muchas partes en movimiento. Contiene toda la maquinaria necesaria para recolectar materiales del ambiente y para construir a partir de ellos una copia de si misma basándose en información hereditaria que contiene (Miller 2006).

Las computadoras nos han familiarizado con el concepto del contenido de información como una cantidad cuantificable. Algunos millones de bytes son necesarios para almacenar este informe, unos cuantos millones más para las fotos de las vacaciones, etc. También nos han familiarizado con el hecho de que la misma información puede ser grabada en diferentes soportes físicos. Como por ejemplo los cassettes que usaban nuestros padres o los CDs y DVDs que son un poco más contemporáneos. Las células, también son capaces de guardar información, y han estado evolucionando y diversificando las

copias de esta información por aproximadamente 3,5 miles de millones de años. Dado que la información que llevan contiene las instrucciones para construir la maquinaria que se encarga de copiar e interpretar dicha información, es poco probable esperar que todas las células guarden la información de la misma manera. Es decir, que el formato en el que guarda información una célula pueda ser leído por la maquinaria de otra. Sin embargo, es así. Todas las células de la tierra almacenan su información hereditaria en la forma de moléculas de ácido desoxirribonucleico (ADN) doble cadena (Figura 1). Estas consisten en dos hebras largas, sin ramificaciones, que se complementan mutuamente y están formadas por una sucesión constituida en base a cuatro tipos de monómeros posibles (Miller 2006).

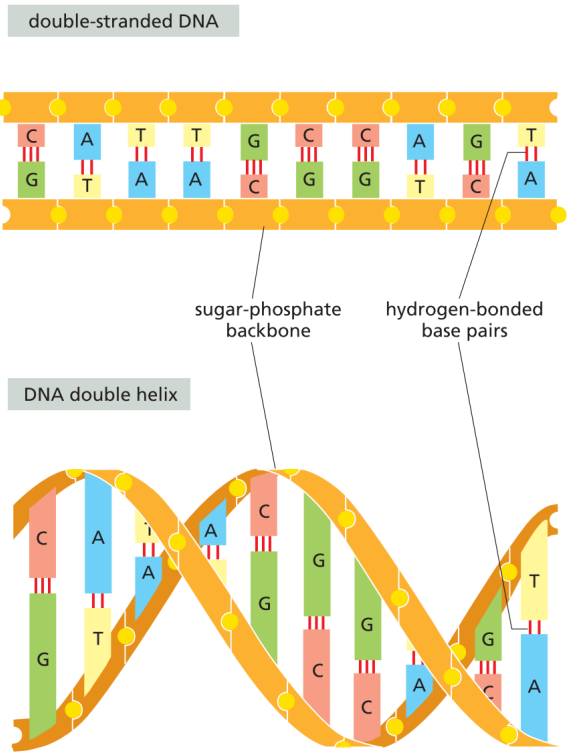


Figura 1: DNA Tomada de Alberts y col. 2017

Estos monómeros, son compuestos químicos conocidos como nucleótidos. Están formados por un azúcar y un grupo fosfato, que son iguales en todos los nucleótidos y les permiten conectarse entre sí mediante enlaces químicos formando una cadena. A su vez, el azúcar se encuentra unido a cada base nitrogenada es el que difiere entre los distintos nucleótidos y le confiere a cada uno su nombre. Puede ser adenina (A), guanina (G), citosina (C) o timina (T). Además, cada base puede aparearse con otra base mediante uniones más débiles conocidas como puentes de hidrógeno. Estas uniones son las que juntan las dos cadenas de ADN y sólo permiten el apareamiento de A con T y C con G, debido a que el número y la orientación de los átomos que pueden formar los puentes de hidrógeno sólo es compatible entre dichos pares (Figura 2).

La información contenida en el ADN está codificada por la secuencia de nucleótidos de una hebra, de la misma forma que la información en una computadora está codificada como una secuencia de unos y ceros. Dado que las hebras se aparean de forma determinística y unívoca, el contenido de información de ambas cadenas es redundante. Como la maquinaria para copiar y leer la información del ADN se encuentra sorprendentemente conservada a lo largo del árbol de la vida, podemos tomar ADN de una célula humana e insertarlo en una bacteria y la información que dicho ADN contiene va a seguir siendo leída, interpretada y copiada exitosamente. De hecho, de esta forma se sintetiza la insulina humana que

se suministra los pacientes diabéticos (Johnson 1983).

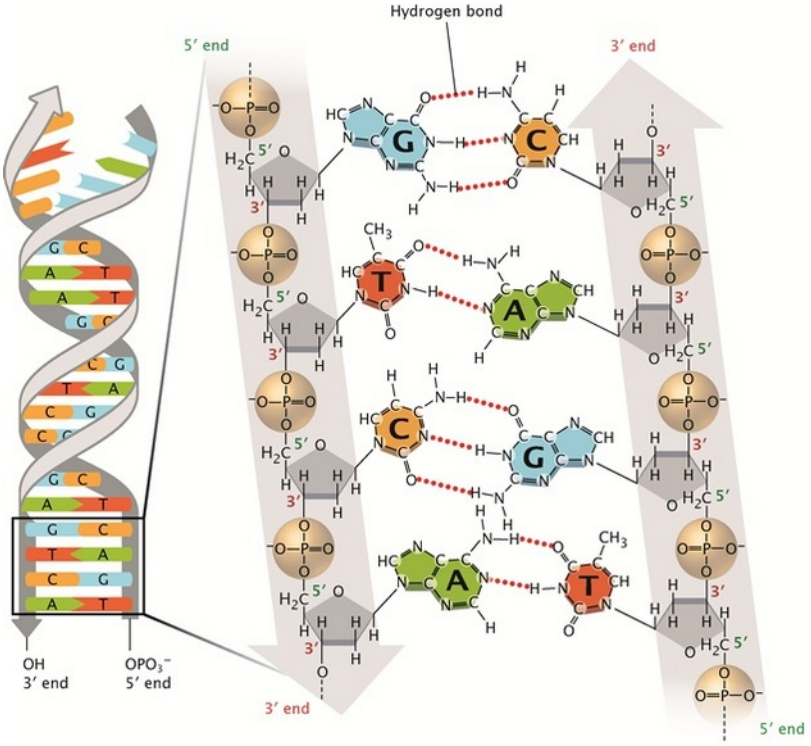


Figura 2: DNA Tomado de Pray 2008. Puentes de hidrógeno líneas rojas punteadas.

Toda la vida del planeta contiene información fundamentalmente en tres tipos de moléculas: ADN, ARN y proteínas. Nos centraremos en el ADN porque es el soporte en el cual las células guardan y transfieren la información de su funcionamiento. Sin embargo, el ARN es empleado para transferir subconjuntos de esta información a diferentes lugares de la célula en los cuales es usado como molde para sintetizar proteínas. Las proteínas forman enzimas que, en última instancia, son las que realizan las acciones en la célula. Facilitan reacciones químicas, envían señales a otras células y son los componentes que dan forma a las estructuras del cuerpo, por ejemplo la queratina de la piel (Miller 2006). En este sentido, se podría pensar al ADN como análogo al esquema de un circuito electrónico complejo, el ARN como el detalle de un componente específico del circuito por ejemplo un circuito integrado, y la proteína como el circuito integrado propiamente dicho.

El conjunto de toda la información contenida en el ADN de un organismo es llamado genoma y especifica todas las moléculas de ARN y proteínas que ese organismo va sintetizar. Se llama gen a cada unidad de información contenida en el genoma que codifica un producto génico (ARN que dará lugar a proteínas u otros tipos de ARN). El genoma humano, por ejemplo, contiene aproximadamente 21000 genes que codifican proteínas y 3100 millones de bases por hebra de ADN (Alberts y col. 2017). La cantidad de bases en una secuencia de ADN se suele representar con la unidad “pares de bases” o pb, haciendo referencia a que el mismo número de bases reside en la cadena opuesta, y se suelen usar prefijos como los del SI (por ejemplo, 3100 Mpb en el caso del genoma humano).

1.2. Alineamiento de secuencias de ADN

El estudio y comparación de secuencias de caracteres derivados de un alfabeto finito es relevante para varias áreas de la ciencia. Es especialmente importante en la biología molecular, dónde ha sido

empleado en el estudio de la evolución de genomas y genes, y el estudio de la estructura y función de proteínas. En los últimos años se realizaron varios proyectos de secuenciación de genomas de distintos organismos, es decir determinar el orden exacto de las bases de su ADN, incluyendo el proyecto genoma humano. Esto generó una cantidad masiva de secuencias de genes y proteínas que requieren algoritmos bioinformáticos para su análisis.

En particular, el alineamiento de secuencias es una forma de acomodar las secuencias de ADN, ARN o proteínas para identificar regiones de similitud que pueden ser consecuencia de relaciones funcionales, estructurales o evolutivas. Permite encontrar relaciones entre las mismas proporcionando un mejor entendimiento de su función, su homología (origen evolutivo común) y revelar regiones conservadas evolutivamente. Si dos secuencias alineadas comparten un ancestro común, las diferencias entre ellas pueden ser interpretadas como mutaciones, inserciones o deleciones que fueron introducidas en uno o en ambos linajes a partir del momento de su divergencia. Por el contrario, la conservación de regiones puede indicar que existe presión selectiva sobre las mismas debida a su rol funcional o estructural.

El alineamiento de secuencias también es una parte fundamental del proceso de secuenciación de genomas. Las técnicas que se usan actualmente para determinar la secuencia de un genoma particular generan en paralelo miles de millones de subsecuencias de tamaño corto, denominadas lecturas, que se encuentran en el rango de 50 a 300 pb[1]. Si el genoma a determinar es de una especie para la cual ya existe un genoma de referencia, entonces las subsecuencias obtenidas deberán ser alineadas al genoma de referencia para poder reconstruir la secuencia del nuevo genoma. En el caso de que no exista un genoma de referencia para la especie que se está secuenciando, es necesario usar otro tipo de algoritmos que escapan el alcance de este trabajo práctico.

Más precisamente, alinear dos secuencias es mapear sus caracteres uno a uno preservando su orden, pero permitiendo la introducción de huecos sin mapear (gaps). Por este motivo, en el caso del ADN es necesario ampliar el alfabeto de 4 letras (A, C, T y G) para incluir un carácter que represente los gaps. Este carácter suele ser el guión medio. En consecuencia, cada carácter de una secuencia puede ser mapeado a un carácter igual en la otra secuencia (match), a un carácter diferente (mismatch) o al carácter de gap. A su vez, cada una de estas posibilidades tiene un puntaje asociado y cada alineamiento tiene un puntaje total dado por la cantidad de caracteres que cae en cada uno de esos casos multiplicado por el puntaje asignado a dicho caso. Se presenta un ejemplo en la Figura 3.

```
ACA - A    (Secuencia 1)
| |   |
ACTGA      (Secuencia 2)

Puntaje total: 1 = 3*1 + 1*(-1) + 1*(-1)
```

Figura 3: Alineamiento de dos secuencias: ACAA y ACTGA. Los puntajes asociados son: match = 1, mismatch = -1 y gap = -1. Los guiones representan los gaps, y las líneas verticales que conectan los caracteres representan los matches.

Hay dos tipos principales de alineamientos denominados local y global. La versión global genera un alineamiento que abarca la longitud completa de las secuencias y es empleado para secuencias de largo similar que se sospechan relacionadas evolutivamente, es decir que se espera que sean relativamente similares. El algoritmo de alineamiento global fue desarrollado por Needleman y Wunsch y normalmente se lo denomina por el nombre de sus autores (Needleman y Wunsch 1970). Por otro lado, el alineamiento de tipo local fue desarrollado por Smith y Waterman y también se lo llama por el

[1]Para más detalle del funcionamiento de una de las técnicas más usadas ver https://www.youtube.com/watch?v=fCd6B5HRaZ8 o https://www.youtube.com/watch?v=CZeN-IgjYCo.

nombre de los autores (Smith y Waterman 1981). Su objetivo es identificar regiones altamente conservadas (similares) a pesar de que el resto de las secuencias pueda presentar gran variación. Una de las motivaciones para el desarrollo del alineamiento local fue la dificultad de obtener alineamientos correctos en regiones altamente similares que se encuentran en un contexto muy variable empleando el alineamiento global. Esta situación es frecuente cuando se comparan secuencias de organismos que están distantes evolutivamente.

1.3. Algoritmos de programación dinámica para el alineamiento de secuencias

Más allá del tipo de alineamiento, es posible encontrar un alineamiento óptimo para cualquier par de secuencias $s1[1...n]$ y $s2[1...m]$. El problema radica en que, para lograrlo, se debe considerar para todo par de posiciones $i < n$ y $j < m$ las 3 opciones posibles de puntaje:

- Asignar un match o un mismatch (son excluyentes dependiendo de si $s1[i] = s2[j]$).
- Asignar un gap en $s1$.
- Asignar un gap en $s2$.

Considerar asignar un gap teniendo la posibilidad de asignar un match puede resultar anti-intuitivo, pero es necesario porque puede llevar a un mejor alineamiento final al seguir avanzando en las secuencias. En consecuencia, el espacio de búsqueda es exponencial respecto del tamaño de $s1$ y $s2$, y el algoritmo resultante de la enumeración de estas posibilidades es sumamente ineficiente.

Este problema puede ser pensado alternativamente de una forma recursiva. Para obtener el puntaje óptimo hasta las posiciones i y j de las secuencias se puede razonar cada uno de los tres casos anteriores en función de un subproblema de menor tamaño que ya ha sido solucionado de forma óptima. Para ello se definen:

- $C_{i,j}$ como el puntaje de asignar un match o mismatch en las posiciones i y j, que puede ser calculado como el puntaje del alineamiento óptimo de las subsecuencias $s1[1...i-1]$ y $s2[1...j-1]$ sumado al puntaje de match si $s1[i] = s2[j]$ o al puntaje de mismatch en el caso contrario.
- $G1_{i,j}$ como el puntaje de insertar una deleción en $s1[i]$, representada por el carácter de gap, alineada con $s2[j]$. Esto puede ser calculado como el puntaje del alineamiento óptimo de las subsecuencias $s1[1...i]$ y $s2[1...j-1]$ sumado al puntaje de gap.
- $G2_{i,j}$ como el puntaje de insertar una deleción en $s2[j]$, representada por el carácter de gap, alineada con $s1[i]$. Esto puede ser calculado como el puntaje del alineamiento óptimo de las subsecuencias $s1[1...i-1]$ y $s2[1...j]$ sumado al puntaje de gap.

Entonces, el puntaje hasta las posiciones i y j se puede definir como el máximo de los puntajes obtenidos en estos tres casos. Es decir, $P_{i,j} = max(C_{i,j}, G1_{i,j}, G2_{i,j})$. Por como fueron definidos los casos, es claro que $C_{i,j}$ depende del subproblema $P_{i-1,j-1}$, $G1_{i,j}$ del subproblema $P_{i,j-1}$, y $G2_{i,j}$ del subproblema $P_{i-1,j}$. Cada uno de estos subproblemas está definido por dos posiciones. Luego, es natural repensar el problema utilizando una matriz como la que se muestra en la Figura 4. Donde cada dimensión de la matriz corresponde a una de las secuencias y cada celda contiene el resultado del subproblema asociado a sus coordenadas.

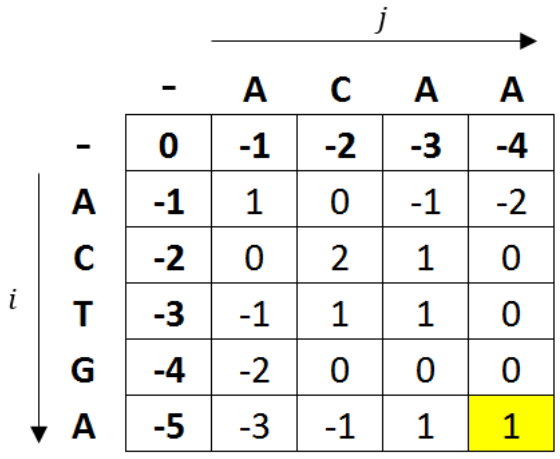


Figura 4: Matriz de programación dinámica para el algoritmo Needleman-Wunsch con un puntaje para match de 1, mismatch de -1 y gap de -1. En amarillo se indica la celda que contiene el puntaje final del alineamiento óptimo. Las puntajes en negrita son los casos base que deben ser inicializados antes de recorrer la matriz.

Sin embargo, para obtener los puntajes de cada celda, necesitamos resolver otros tres subproblemas. A su vez, para cada uno de estos tres se puede aplicar el mismo razonamiento hasta llegar al caso base. Esto nos da una complejidad perteneciente al orden 3^{n+m} de llamados, ya que si se parte desde la coordenada $C_{n,m}$, a lo sumo puede haber $n + m$ pasos hasta la celda $C_{1,1}$, que equivale al caso base. Si bien la cantidad de llamados es del orden 3^{n+m} el número de posiciones en la matriz es $n * m$, lo cual evidencia que se está resolviendo cada subproblema más de una vez. Por lo tanto, si en vez de resolver los subproblemas desde cero cada vez que son encontrados se guarda su resultado la primera vez, se puede obtener en $O(1)$ las siguientes veces, evitando llamados recursivos innecesarios. Esta estrategia se conoce como programación dinámica y permite reducir notablemente la complejidad del algoritmo.

Al plantear el problema de esta forma, es necesario agregar el carácter de gap al principio de cada secuencia para poder contemplar la posibilidad de comenzar el alineamiento con un gap en alguna de las secuencias, y llenar la primera fila y columna con los casos base que se derivan de esta situación. Teniendo en cuenta la dependencia entre los subproblemas mencionada anteriormente, la matriz puede ser llenada de izquierda a derecha y de arriba hacia abajo de forma iterativa a partir de los casos base. Este enfoque se conoce como programación dinámica bottom-up. Una vez que se completan los valores de la matriz el puntaje final del alineamiento óptimo, que es la solución al problema, queda almacenado en la esquina inferior derecha en el caso del alineamiento global (Needleman-Wunsch) y en la celda de valor máximo en el alineamiento local (Smith-Waterman). A continuación se presenta el pseudocódigo de este algoritmo.

Algorithm 1 Algoritmo de alineamiento de secuencias utilizando programación dinámica bottom-up.

```
1: function alinear(s1[1...n], s2[1...m])
2:     Inicializar la primer fila de la matriz de programación dinámica M
3:     Inicializar la primer columna de la matriz de programación dinámica M
4:     for i: 1 to n do
5:         for j: 1 to m do
6:             C_{i,j} ← M_{i-1,j-1} + W(i,j)
7:             G1_{i,j} ← M_{i,j-1} + G
8:             G2_{i,j} ← M_{i-1,j} + G
9:             M_{i,j} ←max(C_{i,j}, G1_{i,j}, G2_{i,j})
```

En el caso de que el alineamiento sea local (Smith-Waterman), inicializar la primer fila y la primer columna de la matriz de programación dinámica implica llenar cada posición con cero. En el caso de que el alineamiento sea global (Needlman-Wunsch), se debe sumar el costo del gap una vez por cada posición a medida que se avanza. Adicionalmente, si el alineamiento es local, no se deben permitir guardar valores negativos en la matriz. Los mismos deberán reemplazarse por cero.

Dado que la inicialización de la matriz es $O(nm)$ para su creación y $O(n) + O(m)$ para sus casos bases, y los dos loops anidados dan complejidad de $O(nm)$, ambos tipos de alineamiento tienen una complejidad total de $O(nm)$. Es decir que la estrategia de programación dinámica permite pasar de una implementación ingenua de orden exponencial a una que es cuadrática respecto del tamaño de la secuencia de mayor largo.

Una vez encontrado el puntaje óptimo, es necesario reconstruir el alineamiento correspondiente. Para ello es necesario recorrer la matriz de programación dinámica a partir de la celda que contiene el óptimo e ir armando el alineamiento de atrás hacia adelante, por este motivo esta parte del algoritmo se denomina *backtracking*. En la figura 5, se muestran las diferencias en el backtracking entre un alineamiento global y uno local de las mismas secuencias.

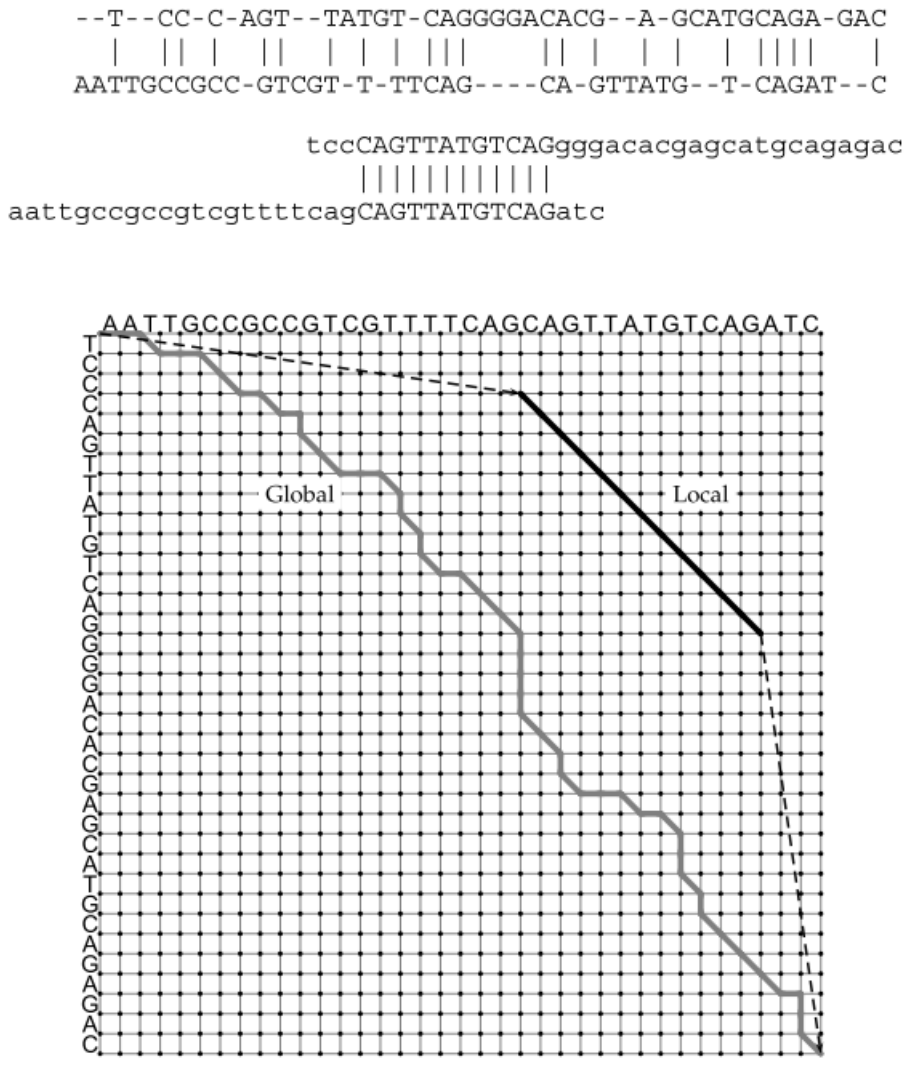


Figura 5: Caminos de reconstrucción del alineamiento de dos secuencias a partir de la matriz de programación dinámica para la versión local y global del algoritmo. Arriba se muestran los alineamientos resultantes, primero el global y luego el local. Las letras en mayúscula son las que se incluyeron en el alineamiento, los guiones representan los gaps, y las líneas verticales que las conectan representan los matches. Tomada de (Miller 2006).

En el caso del alineamiento global el backtracking comienza desde la esquina inferior derecha de la matriz mientras que, en el caso del alineamiento local, comienza desde la posición que tiene el máximo puntaje. Luego, se debe determinar de cuál de las tres decisiones posibles que se toman al llenar la matriz proviene el puntaje de la celda actual, comparándola con las celdas que están inmediatamente a su izquierda, arriba y en la diagonal superior izquierda:

- Si el puntaje viene de asignar un match o un mismatch, entonces se colocan en el alineamiento óptimo las bases correspondientes y se avanza en diagonal hacia arriba y a la izquierda una posición.
- Si el puntaje viene de asignar un gap, entonces se coloca el gap en la secuencia indicada y la base correspondiente en la otra secuencia. Luego, se avanza hacia la izquierda si se colocó el gap en la secuencia izquierda y hacia arriba si se colocó el gap en la secuencia superior.

La reconstrucción del alineamiento termina, en el caso del método global, cuando se llega a la esquina superior izquierda de la matriz. En el caso de alineamiento local, se termina cuando se llega

a una celda cuyo puntaje es cero. En consecuencia, el backtracing es en el peor caso $O(n) + O(m)$. Por lo tanto la complejidad del alineamiento de dos secuencias, aún incluyendo la reconstrucción del alineamiento óptimo, es $O(nm)$. Sin embargo, se pueden conseguir mejoras en el desempeño de este algoritmo si se paraleliza el llenado de la matriz. Para ello, una opción es utilizar el modelo de ejecución Single instruction multiple data (SIMD).

1.4. Modelo de ejecución Single instruction multiple data (SIMD)

Es un modelo de ejecución capaz de computar una sola operación sobre un conjunto de múltiples datos. Resulta particularmente útil para procesar audio, vídeo, o imágenes en donde se aplican algoritmos repetitivos sobre conjuntos de datos del mismo formato y que se procesan en conjunto, como es en nuestro caso para el alineamiento de secuencias. Para utilizar este modelo de ejecución utilizamos distintos conjuntos de instrucciones.

1.4.1. *GCC* Compiler intrinsic function

GNU Compiler collection (GCC) ofrece un punto intermedio entre *assembly* y C, esto nos brinda un incremento en la velocidad de procesamiento y algunas características del procesador sin tener que ir directamente al lenguaje ensamblador. Las *Compiler intrinsics* a diferencia de las funciones de biblioteca, son generadas en tiempos de compilación. Además exponen la funcionalidad del procesador de manera específica, por lo cual se obtiene una funcionalidad similar a la de assembly pero delegando algunas funcionalidades al compilador como el chequeo de tipos, almacenamiento de registros y scheduling de instrucciones.

En el contexto de este trabajo aprovechamos este tipo de funciones para mejorar el desarrollo de los algoritmos, usando SIMD intrinsics. La sintaxis que poseen es similar a un llamado de función, pero generalmente produce una sola instrucción a nivel assembly. Esto nos trae un incremento significativo de performance y además facilita el desarrollo en general.

1.4.2. Streaming SIMD extensions (SSE)

SSE introduce la idea de tener una sola instrucción que procese múltiples datos a la vez. Para esto agrega nuevos registros de 128 bits, nombrados XMM0-XMM15, y a su vez instrucciones para operar con estos. Permite hacer operaciones entre datos de distintos tamaños, como byte, word, float, double, etc.

Si bien inicialmente estaba SSE, se fueron agregando distintas extensiones, como SSE2, SSE3, SSE4, etc. La primera de todas (SSE) introdujo operaciones que procesaban múltiples números de punto flotante simultáneamente, y mas adelante se incluyo soporte para números enteros.

1.4.3. Advanced vector extensions (AVX)

Estas extensiones vienen en dos partes, AVX y AVX2, donde la primera introduce operaciones para punto flotante y la otra las equivalentes para enteros. A diferencia de SSE, en AVX hay 2 cambios notorios, a nivel registro y en formato de instrucciones.

- Por un lado, los registros de 128 bits que venian de SSE se incrementan en ancho a 256 bits en AVX, y pasan a llamarse YMM0-YMM15.
- El otro aspecto que cambia es que AVX introduce un nuevo formato, de 3 operandos. Tanto el destino como los dos operandos fuentes pasan a ser registros distinguidos. Esto permite no perder el contenido de los registros con los que se opera, y evitar hacer copias de los valores para no perderlos.

A continuación se ve un ejemplo de este formato, con la instrucción suma en versión SSE y en AVX:

1. SSE: **paddw xmm1, xmm2**
2. AVX: **vpaddw ymm1, ymm2, ymm3**

En el primer caso la suma de xmm1 con xmm2 se guarda en xmm1, perdiendo el contenido de este. En cambio, en la segunda la suma de ymm2 con ymm3 se guarda en ymm1, preservando los operandos fuente.

1.4.4. Advanced vector extensions of 512 bits (AVX-512)

Finalmente, en AVX512 se tienen varias extensiones del set de instrucciones hasta el momento, cada uno con su identificación particular. Todas estas comparten varios aspectos nuevos, como tamaños de registros, nuevos registros y formato de instrucción.

Los registros en este caso se extienden de 256 bits a 512 bits, y además se agregan 16 extra sobre los 16 existentes hasta el momento. A estos se los llama ZMM0-ZMM31, y se puede acceder a su parte YMM (256 bits) o XMM (128 bits) para utilizarlo en las instrucciones existentes de SSE y AVX. También se agregan nuevos registros de 64 bits, llamados mask registers y referenciados como k0-k7. Estos tienen sus propias instrucciones para manipularlos, y están involucrados en las nuevas instrucciones de AVX512.

Las instrucciones de SSE y AVX presentan nuevas versiones en esta extensión, introduciendo features como **opmask**, mas registros involucrados, una opción booleana para cambiar el comportamiento de la instrucción, etc. Para poner un ejemplo de esto, tomamos como ejemplo la instruccion add a nivel word, pero en su version 512. Esta instrucción puede utilizarse de distintas formas, dependiendo de si se especifica o no los parametros {k} y {z}:

- Si no se especifica ningun parametro:

  vpaddw zmm1, zmm2, zmm3

  La instruccion suma word a word entre zmm2 y zmm3, y guarda los resultados en zmm1
- Si se especifica un valor para el registro mascara {k} entre k1-k7:

  vpaddw zmm1 {k}, zmm2, zmm3

  Cada suma se guarda en su respectivo lugar en zmm1 si el bit correspondiente en la mascara k esta en 1, sino se preserva el valor en zmm1.

- Finalmente, si se utiliza la {z}:

  vpaddw zmm1 {k}{z}, zmm2, zmm3

  Se comporta casi igual que en el caso anterior, solo que en lugar de preservar el valor en zmm1 si el bit es 0, lo pone en 0.

1.5. Intel software development emulator (SDE)

Al desarrollar código utilizando SIMD, se tienen múltiples extensiones del set de instrucciones, como los mencionados previamente, pero no todas están disponibles en la arquitectura del procesador que se tiene.

Debido a esto es necesario, de alguna forma, ëmularl̈as instrucciones que el procesador no conoce, para así poder ejecutar el código que se está desarrollando. Este es el objetivo de la herramienta SDE desarrollada por Intel, el de poder probar las nuevas extensiones, sin necesidad de tener el procesador adecuado. Esta herramienta permite seleccionar la arquitectura que se quiere emular, para poder llevar la simulación lo mas cercano posible a estar utilizando un procesador real que si soporta las instrucciones correspondientes.

1.6. Formato FASTA de archivos

En bioinformática y bioquímica, el formato FASTA es un formato de texto para representar secuencias de nucleótidos o secuencias de aminoácidos (proteínas), en el que los nucleótidos o aminoácidos se representan mediante códigos de una sola letra. El formato se originó en el paquete de software FASTA, pero ahora se ha convertido en un estándar casi universal en el campo de la bioinformática.

Una secuencia en formato FASTA comienza con una descripción de una sola línea (header), seguida de los datos de las secuencia. La línea de descripción (defline) se distingue de los datos de secuencia por el símbolo “>” al principio. Se recomienda que todas las líneas de texto tengan menos de 80 caracteres. Una secuencia de aminoácidos con este tipo de formato sería:

```
>NC_045512.2 | SARS-CoV 2 isolate Wuhan-Hu-1, complete genome
ATTAAAGGTTTATACCTTCCCAGGTAACAAACCAACCAACTTTCGATCTCTTGTAGATCT
GTTCTCTAAACGAACTTTAAAATCTGTGTGGCTGTCACTCGGCTGCATGCTTAGTGCACT
CACGCAGTATAATTAATAACTAATTACTGTCGTTGACAGGACACGAGTAACTCGTCTATC
TTCTGCAGGCTGCTTACGGTTTCGTCCGTGTTGCAGCCGATCATCAGCACATCTAGGTTT
```

Figura 6: Primeras 5 líneas de la secuencia de referencia para SARS-CoV 2 extraida del repositorio NCBI Virus (NCBI)

La simplicidad del formato FASTA facilita la manipulación y el análisis de secuencias. Además, otra ventaja importante son los repositorios o sitios de libre acceso donde se pueden obtener secuencias con este formato.

1.7. Objetivo

El objetivo general de este trabajo práctico es emplear el conocimiento obtenido en la materia sobre el modelo de ejecución SIMD para implementar de forma paralela los algoritmos de alineamiento de secuencias Smith-Waterman y Needleman-Wunsch. Como objetivos particulares, se propone utilizar los conjuntos de instrucciones SSE, AVX y AVX-512, y comparar el desempeño de los distintos conjuntos cuando los mismos son empleados a nivel de lenguaje ensamblador o en lenguaje C mediante el uso de *intrinsics*. Adicionalmente, se evaluará el impacto de utilizar distintos grados de optimización en la compilación del código C y su interacción con el uso de los distintos conjuntos de instrucciones. Finalmente, se pondrá a prueba el desempeño de los algoritmos implementados con secuencias reales provenientes de genomas virales.

2. Desarrollo

A continuación vamos a profundizar en la metodología utilizada para realizar las implementaciones de los algoritmos. Para ello, vamos a explorar los modelos de ejecución, los distintos conjuntos de instrucciones y las herramientas empleadas durante el desarrollo. Posteriormente, abordaremos la metodología experimental.

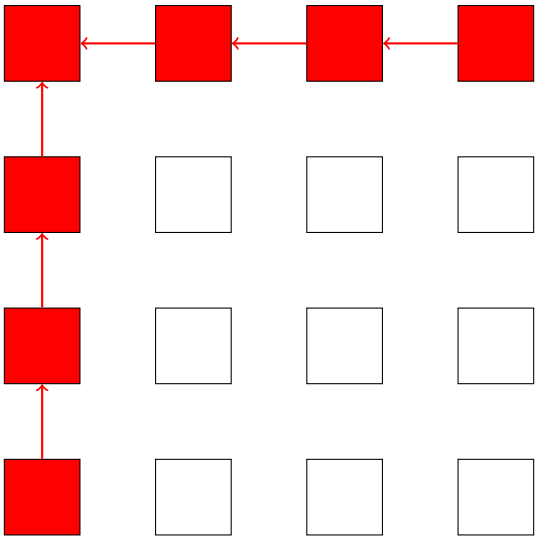
2.1. Metodología de desarrollo

Dada la compleja tarea de paralelizar este tipo de algoritmos, el desarrollo debe ser llevado a cabo de forma gradual y precisa para evitar errores graves. La primer versión implementada de los algoritmos fue realizada con un enfoque "lineal" en *C* de forma que podamos afianzar nuestro conocimiento acerca de los pasos que debe llevar a cabo el algoritmo. Luego llevamos esta implementación a *assembly* donde, si bien el desarrollo fue más complejo, al realizarlo con la lógica tradicional comenzamos a familiarizarnos con el comportamiento del algoritmo a más bajo nivel. A partir de estas implementaciones comenzamos a modificar la naturaleza de los algoritmos para poder paralelizarlos. Primero implementamos en *C* utilizando las *compiler intrinsics* para lograr una traducción mas amena a la hora de realizar la implementación final en *assembly*. Cada paso de extensión del conjunto de instrucciones o cambios en la lógica de los algoritmos fue testeado de manera exhaustiva para reducir la presencia de errores.

Un paso en común para todos los algoritmos es la recuperación de las secuencias alineadas, es decir, *Backtracking*. En particular, este paso no puede ser paralelizado y la recuperación de cada string resulta de orden lineal. Considerando esto, optamos por hacer que todas las versiones de los algoritmos utilicen la misma función de backtracking adaptándose a quien lo esté llamando.

2.1.1. Paralelizacion

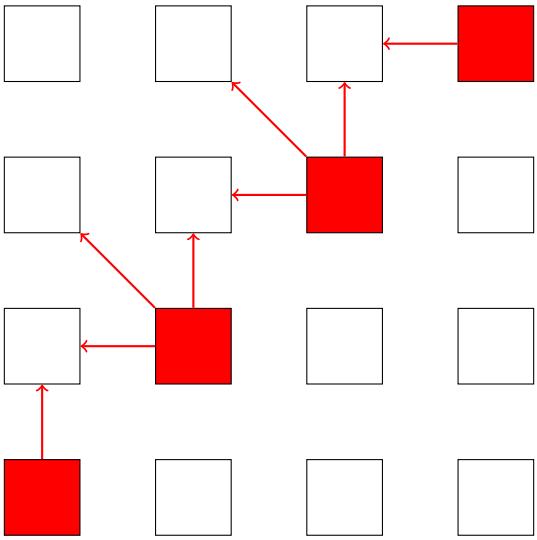
Debido a la dependencia de las celdas de la matriz en ambos algoritmos de alineamiento, no puede aplicarse SIMD sobre esta, suponiendo que las filas se ubican contiguamente en memoria. Para ejemplificar esto, supongamos que tengamos la siguiente matriz:



Podemos observar que, para la primer fila por ejemplo, el valor de cada celda excepto por la primera depende del valor de la anterior. Esto mismo pasa si vemos la primer columna, por lo que no podemos calcular paralelamente todos estos valores, ya que hay una dependencia entre ellos.

Esto parece ser un impedimento para la aplicación de SIMD, y hasta imposible de encontrarle una solución. Sin embargo, es posible superar este problema, viendolo desde un enfoque distinto y poco tradicional dentro de las aplicaciónes clasicas de SIMD sobre matrices.

La idea de este nuevo enfoque es concentrarse en las diagonales de la matriz, en lugar de las filas y columnas. Una vez mas, si observamos el ejemplo anterior:

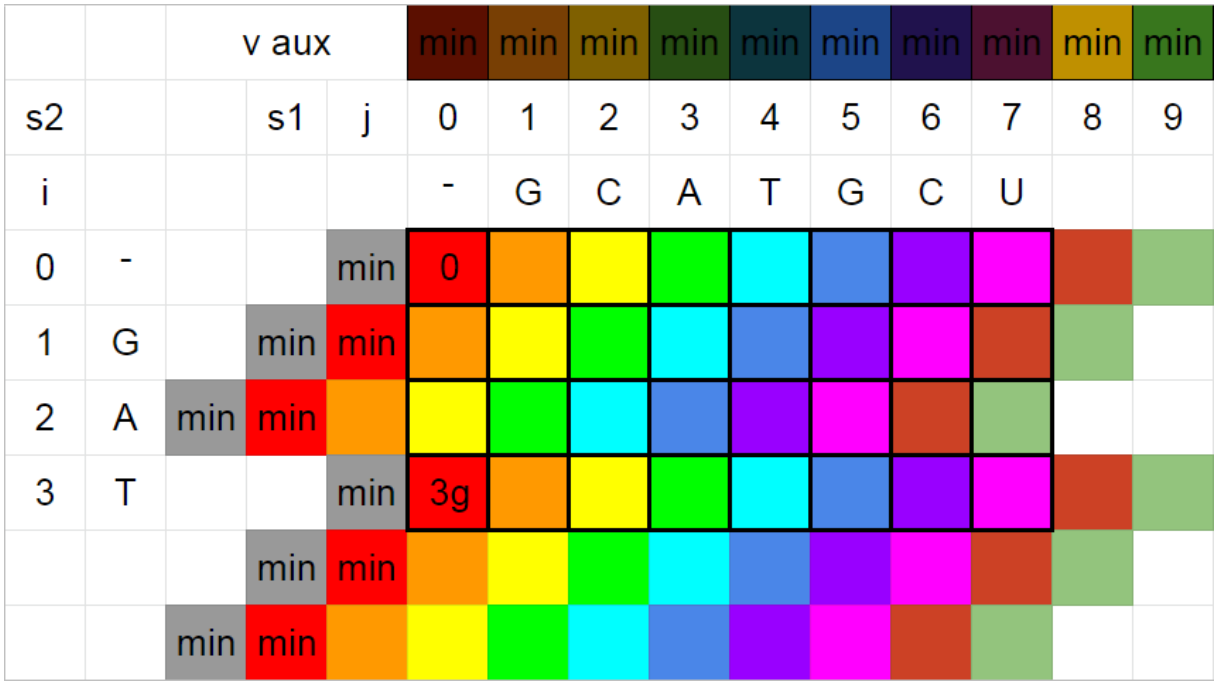


Aqui vemos que para toda celda perteneciente a la misma diagonal, su valor no depende de ninguna de las otras. Esto implica que se puede procesar cada celda de esta diagonal simultaneamente, sin generar conflictos de dependencias.

Una vez definida la forma de procesar los valores de la matriz, hace falta ver en que orden y como representar en memoria esto. Por un lado, el orden esta determinado por la dependencia de los calculos, el cual obliga a procesar las diagonales de izquierda a derecha. Respecto a forma de representar esto

en memoria, también surge naturalmente que la forma correcta es guardar contiguamente en memoria las diagonales de izquierda a derecha. Sin embargo, el largo que tienen estas diagonales en memoria es conveniente fijarlo basado en cuantas celdas pueden procesarse en paralelo. Una última observación necesaria para terminar con este modelado es que, hay casos donde en el procesamiento se requieren los valores de las celdas superiores, los cuales pueden no existir o ser dificiles de ubicar en memoria. Por estas razones, definimos un vector auxiliar, el cual posee los valores de la fila necesaria para cubrir este problema.

A continuación presentamos el modelo final reuniendo todas estas ideas, bajo un ejemplo con 2 secuencias arbitrarias y suponiendo que se está en el caso de alineamiento global:



Donde:

- La secuencia 1 es “GCATGCU”
- La secuencia 2 es “GAT”
- Para este caso pueden procesarse 3 celdas a la vez
- El valor “min” representa el valor mas pequeño dentro de la representación entera que se utilice.
- Cada diagonal esta denotada con un color distinto
- La letra g hace referencia a la penalidad del gap.

En el modelo propuesto, la matriz termina dividida a nivel filas en lo que llamaremos **franjas** a lo largo del trabajo. Cada franja consiste de una cantidad de diagonales, y la cantidad de franjas y diagonales por franja esta determinada por los siguientes calculos:

$$\#franjas = \frac{(longS2 + longDiagonal - 1)}{longDiagonal}$$

$$\#diagonales = longS1 + longDiagonal$$

Notar que no todas las celdas de estas diagonales forman parte de la matriz, sino solo las que están remarcadas con recuadro negro. Si bien esto es un desperdicio de memoria, es la mejor forma que encontramos para representar al modelo.

Por otro lado, las primeras 2 diagonales de cada franja representan los casos base del algoritmo. A diferencia de la versión lineal, solo hace falta inicializar algunas celdas de la primer columna de la matriz, ya que el resto se procesara solo, con la ayuda del vector auxiliar llamado **v aux** en la imagen. Además, las celdas que no se encuentran en la matriz real tendrán asignado un valor adecuado para no afectar los cálculos sobre las celdas reales y generar valores erróneos.

Resta ver en que consisten los pasos de una iteración del algoritmo en este nuevo modelo. Podemos dividirlo en estos pasos:

- **Paso 1:** Obtener y ubicar correctamente los caracteres necesarios de s2 para la iteración.
- **Paso 2:** Obtener y ubicar correctamente los caracteres necesarios de s1 para la iteración.
- **Paso 3:** Obtener y ubicar correctamente los valores necesarios de las diagonales para la iteración. Esto a su vez se divide en tres, una por cada dirección de donde puede obtenerse el valor:
  - Diagonal con valores a la izquierda de la actual
  - Diagonal con valores arriba de la actual
  - Diagonal con valores diagonales a la actual
- **Paso 4:** Tomar el máximo entre las tres direcciones para cada valor
- **Paso 5:** Actualizar la matriz y el vector auxiliar adecuadamente
- **Paso 6:** Actualizar la mejor posición de la matriz de un alineamiento local (solo en Smith Waterman).

Pasamos ahora a explicar en detalle cada uno de estos pasos:

Paso 1

La subsecuencia de **s2** necesaria para una iteración se corresponde con la franja en la que está ubicado en ese momento. Por ejemplo, para el ejemplo anterior, en la primera franja la subsecuencia asociada es **-GA**. En cambio, para la segunda franja, la subsecuencia es **TXX**, donde X indica que no existe un carácter en esta posición, ya que s2 no es lo suficientemente largo. Finalmente, como consecuencia de la ubicación en memoria de las diagonales y de **s2**, es necesario invertir la subsecuencia, para que los cálculos sean correctos mas adelante.

Paso 2

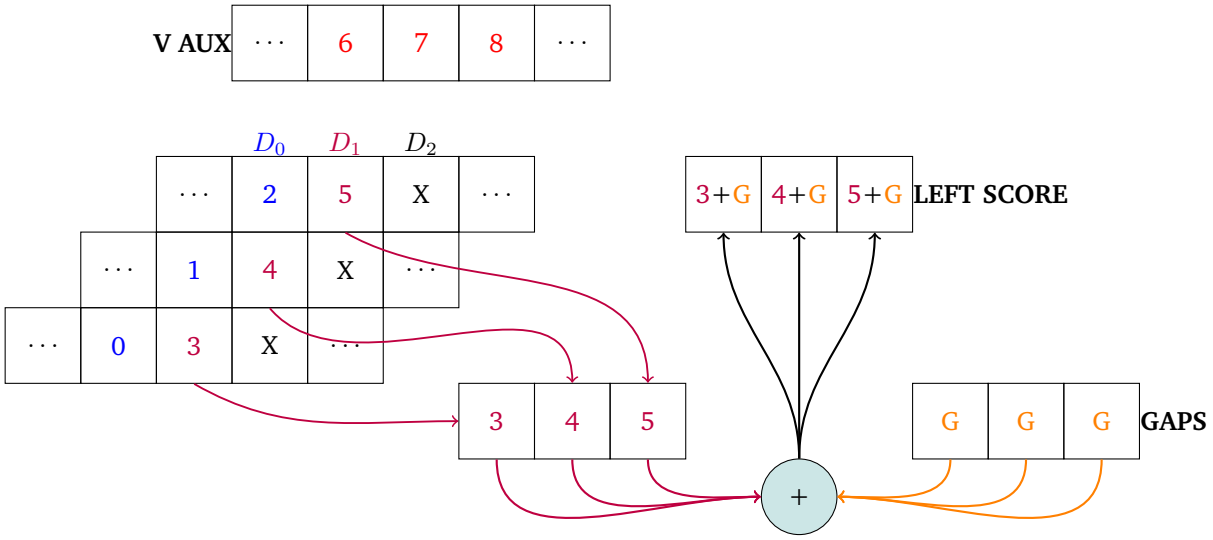
El procedimiento en este caso es similar al paso 1, solo que la subsecuencia asociada es la que se alinea con las celdas de la diagonal que se quiere calcular, y no es necesario invertirla posteriormente.

Paso 3

En esta parte se quiere obtener, para cada celda a procesar, los valores de las celdas a izquierda, arriba y diagonal de cada una. A continuación se puede visualizar cada uno de estos casos, y como se obtienen los respectivos valores en cada uno. En todos los esquemas se considera la franja actual de la iteración, y se indica como D_0 y D_1 a las diagonales anteriores a la que se quiere calcular en ese momento, D_2. Se indica el vector auxiliar **V AUX** que se mencionó previamente, y dependiendo el caso se tiene un registro **GAPS** o **CMPS**. El primero tiene en sus posiciones la penalización del gap, y el segundo tiene en cada posición el puntaje de match o missmatch dependiendo de si los caracteres de $s1$ y $s2$ coinciden en esa posición. Finalmente, para cada caso se tiene un registro **"DIRECCION" SCORE** que contiene los valores finales de los cálculos de venir por una dirección. Finalmente, a cada celda de las diagonales y del vector auxiliar se les asigno un numero para identificarlas, y seguir de manera mas sencilla el procedimiento.

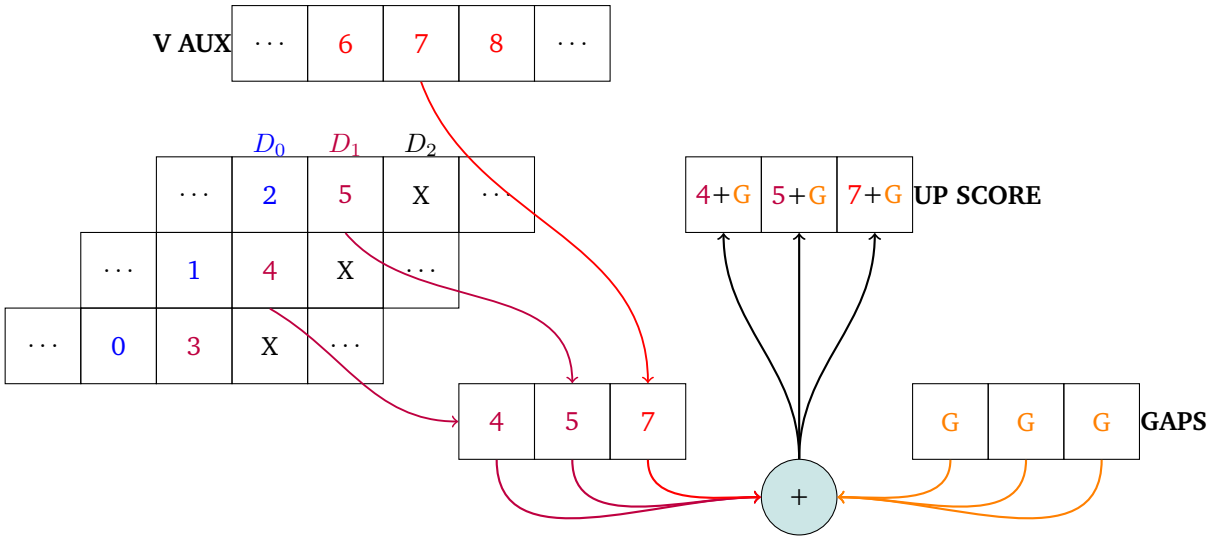
Dirección izquierda

En el caso de venir por izquierda, se necesitan las celdas que son adyacentes a D_2 a izquierda, es decir, D_1. Luego de tener a esta, lo que resta hacer es sumarle la penalización de gap en cada posición, y se obtiene el puntaje final para cada posición.



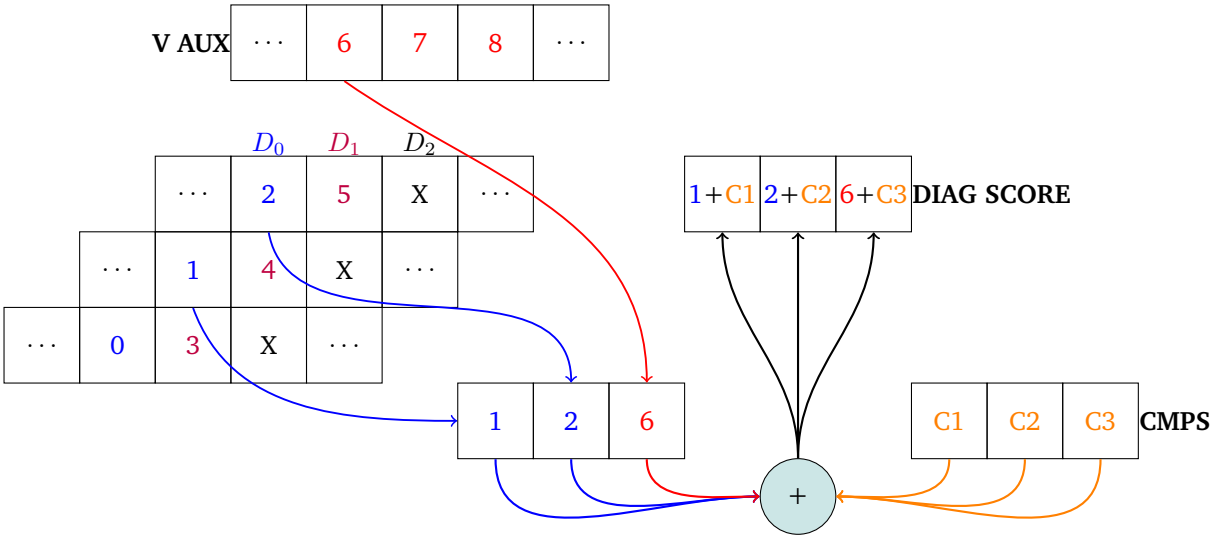
Dirección arriba

Cuando se viene por la dirección superior, una vez mas se necesitan los valores adyacentes a D_2 directamente arriba de esta. A diferencia del caso anterior, D_1 no tiene todos los valores necesarios, sino que es necesario consultar el valor de **V AUX** que se encuentra alineado verticalmente con la celda de D_2 en la posiciónposición mas alta, es decir la 7. Una vez obtenido esto, se le suma nuevamente como en el caso anterior a cada posición la penalización.



Dirección diagonal

Finalmente, calcular los valores de provenir diagonalmente es similar al caso de provenir desde arriba. La primer diferencia es que se extraen valores de D_0, y se obtiene el ultimo valor de la celda 6 en **V AUX**, la que es adyacente diagonalmente a la mas superior de D_2. La segunda y ultima diferencia es que, en lugar de sumar el gap en cada posición, se suma el puntaje de match o missmatch según corresponda, utilizando el registro **CMPS**, como indica el paso del algoritmo en este caso.



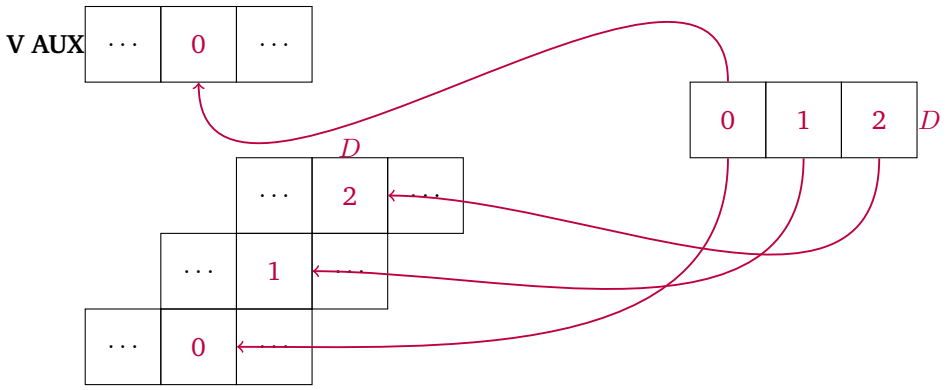
Paso 4

Realizar este paso es simple, obtener el máximo entre las tres direcciones posibles para cada celda de la diagonal actual En el caso de tratarse de un alineamiento local, se toma el máximo entre las tres anteriores mas el valor 0 para cada celda.

Paso 5

Actualizar la matriz consiste simplemente en guardar la diagonal en la posición correcta en memoria.

En el caso del vector auxiliar, actualizarlo implica guardar el valor que se encuentra en la celda mas baja de la diagonal en la del vector auxiliar que se encuentra alineada con esta. Estos pasos se pueden visualizar a continuación:



Paso 6

Este último paso solo se lleva a cabo en alineamiento local, en el algoritmo de Smith Waterman. Consiste en chequear si se encontró un valor superior al óptimo hasta ahora dentro de los que se calcularon en la iteración actual, y en tal caso actualizar el óptimo por ese.

Simulación de SIMD en C

A la hora de realizar una implementación en SIMD, hay que ser muy cuidadoso con todas las manipulaciones que se hagan de los datos. El diseño del algoritmo en SIMD puede llevar a muchas complicaciones, y volverse complejo de verificar su correctitud. Por estos motivos decidimos, en el momento de desarrollar las implementaciones, hacer primero una implementación sin aplicar instrucciones SIMD, *simulando* cada una en C.

La simulación de una instrucción es directa, los registros SIMD pasan a ser arrays del tipo de dato y longitud apropiado, y las instrucciones se realizan con un loop sobre estos arrays, posición a posición.

Lo que permite esto es poder fijar una longitud arbitraria para los registros, y todo se adapta a esa longitud. Al hacer esto, se puede simular código SIMD para los tamaños de registros deseados, en este caso es 128 (SSE), 256 (AVX) y 512 (AVX-512) bits. Para ejemplificar esto, supongamos que queremos simular la suma de dos registros **A** y **B** con un tamaño arbitrario, y guardarlo en uno **C**:

```
// Suma dos registros A y B y guarda el resultado en C
C := A
para i desde 0 hasta len(A) - 1:
    C[i] := C[i] + B[i]
```

La ventaja una vez más de esto último es que funciona para cualquier longitud de registro, otorgando una gran flexibilidad.

2.2. Metodología de Testing

Para asegurar el funcionamiento correcto de nuestras implementaciones tomamos diferentes enfoques aprovechando ciertas propiedades del problema. En la implementación de los casos de test utilizamos la herramienta *liblittletest*[2], la cual facilitó el desarrollo de los mismos de una manera significativa.

- **Test estático:** El primero de estos enfoques consiste en calcular la matriz de puntajes junto con los alineamientos correctos a mano para determinadas secuencias, y luego probar que el algoritmo genere de manera correcta dicha matriz junto con los alineamientos. Este tipo de test se realiza sobre las implementaciones lineales en c para ambos algoritmos, con dos casos distintos de prueba.
- **Test aleatorio :** Para este segundo enfoque generamos de manera aleatoria y con tamaño variable ambas secuencias. A partir de estas utilizamos como "oráculo" el algoritmo lineal en c. Sobre estas secuencias ejecutamos el algoritmo a testear y comparamos que la matriz de puntajes generada coincida posición a posición junto con los alineamientos y el puntaje máximo obtenido. Este método de testing se realizo sobre todos los algoritmos implementados en *C* para un mismo caso, y para cada implementación hecha en *assembly* se genera un test particular. Dada la naturaleza aleatoria de las secuencias implementamos un script en bash, que ejecuta estos tests randomizados para todas las versiones de los algoritmos hasta que alguno falla. De esta manera nos aseguramos un método de testing más exhaustivo.

2.3. Implementación

Al realizar distintas versiones de los algoritmos hay diferentes enfoques en el proceso de desarrollo para cada una, lo cual lleva a consideraciones particulares para remarcar de cada uno. A su vez, hay dos consideraciones común entre todas.

La primera involucra el tipo de dato utilizado para llevar a cabo todos los cálculos de todas las implementaciones presentes a continuación. El tipo utilizado es word con signo, lo que implica un rango de valores de [-32768,32767], y limita la maxima longitud entre ambas secuencias con las que se trabaja dependiendo del valor de gap, match y missmatch. Por ejemplo, para un valor de gap y missmatch de -1 y match de 1, se puede trabajar con secuencias de hasta 32768 caracteres aproximadamente. MENCIONAR QUE ESTO ESTA BASTANTE BIEN CONSIDERANDO EL USO COMUN DE ALINEAMIENTO

La segunda consideración intenta reducir la cantidad de accesos a memoria que se hace a la matriz de puntajes, notando que en cada iteración solamente se necesitan las dos diagonales anteriores a las celdas que se quieren procesar. Esto quiere decir que en todo momento puede tenerse en 2 registros $r1$ y $r2$ las 2 diagonales anteriores a la actual y consultar los valores desde ahi, sin necesidad de acceder a memoria. Luego al finalizar la iteración pueden actualizarse estos 2 registros, copiando el valor de $r1$ en $r2$, y el de la diagonal actual a $r2$, suponiendo que $r1$ sea la primer diagonal a la izquierda de la diagonal actual. Con esta mejora se logra evitar 2 accesos a memoria innecesarios en toda iteración

Una distinción interesante de las implementaciones realizadas para *assembly* es que, para el algoritmo de alineamiento local, utilizamos un registro *XMM* para guardar dentro del mismo la posición

[2]Esta es la dirección asociada al repositorio de la herramienta https://github.com/etr/liblittletest

(x e y) y el valor correspondiente al máximo global obtenido. En la figura 8 se puede observar la disposición de estos 3 valores dentro del *XMM* llamado **BEST X Y GLOBAL**.

Figure 8: Disposición del máximo global y su posición almacenado en un *XMM*

2.3.1. Sin paralelización

La implementación de la versión lineal de los algoritmos se realizo en C, y refleja el comportamiento expresado en el pseudocódigo presentado en la introducción 1. La diferencia entre ambas implementaciones por un lado es que, en el caso de Smith Waterman se debe tomar el máximo entre las 3 variables $C_{i,j}, G1_{i,j}, G2_{i,j}$ y 0. Por el otro, a la hora de hacer el backtracking, en Smith Waterman debe partirse desde la posición que posee el mayor puntaje de toda la matriz. Mientras que para Needleman-Wunsch se mantiene la lógica en el pseudocódigo.

2.3.2. SSE

Esta implementación utiliza la extensión de instrucciones SSE para llevar a cabo la paralelización del algoritmo. Hablar de todos los detalles de esta implementación no resulta muy interesante, pero si vale remarcar ciertos aspectos de los pasos del algoritmo descriptos previamente, que no resultan triviales de realizar.

Paso 1 y 2: Lectura de secuencias **s1** y **s2**

En el paso que involucra leer de memoria la subsecuencia de **s2** necesaria para procesar la última franja puede ocurrir que el tamaño de la secuencia sea menor a la cantidad de caracteres que se quiere leer, provocando accesos a memoria fuera de la secuencia. Esto sucede a su vez en el paso equivalente a este pero que involucra a la secuencia **s1**, cuando se quiere calcular las primeras y las últimas diagonales de una franja. Como estas tienen una porción fuera de la matriz de puntajes real esto implica que, al leer la subsecuencia de **s1** correspondiente para procesar se producen nuevamente accesos a memoria inválidos.

Este problema puede solucionarse en dos simples pasos. Primero para evitar accesos fuera de memoria se puede calcular un *offset* particular para cada caso, que permite leer de memoria dentro de los límites. Sin embargo esta lectura no deja en el registro los caracteres deseados en sus posiciones correspondientes para los cálculos posteriores. El segundo paso se encarga de solucionar este nuevo problema, desplazando en la dirección que corresponda y la cantidad adecuada los caracteres dentro del registro, para luego colocar en los lugares desplazados valores que no nos afecten en las comparaciones a realizar para el cálculo de puntajes. Para el caso de la secuencia **s2** la dirección del desplazamiento es hacia la derecha, mientras que para **s1**, si se trata de las primeras diagonales es a izquierda, y sino a derecha. Finalmente, para evitar que las comparaciones entre caracteres validos e inválidos o entre inválidos e inválidos presenten coincidencias no deseadas, definimos una convención. En el caso de la secuencia **s2**, luego de desplazar los caracteres insertamos valores distintos a 0 y al abecedario que se utiliza (ACGT) en las posiciones que corresponden a posiciones fuera de la memoria de **s2**, y para **s1**

insertamos 0 directamente. Como en ambos casos los valores insertados no coinciden con caracteres validos y tampoco entre ellos, nunca producen coincidencias no deseadas.

Paso 3: Registro CMPS para el score diagonal En el paso 3 mencionado previamente en la sección de paralelización del algoritmo, se utiliza un registro CMPS que contiene en cada posición el puntaje de match o missmatch según coincidan o no los caracteres de **s1** con **s2**. La manera en la que generamos este registro es a través de las instrucciones *compare* y *blend*. Primero hacemos las comparaciones de a pares entre las subsecuencias de **s1** y **s2** de la iteración actual, y dejamos este resultado en el registro *MASK*, como ejemplifica la figura 9 en la primer parte. Luego utilizando este registro como seleccionador para la instrucción blend, en conjunto con los 2 registros *MA* y *MI*, los cuales contienen en todas sus posiciones el puntaje de match y missmatch respectivamente, podemos generar el registro *CMPS* deseado, que se visualiza en la segunda parte de la figura 9.

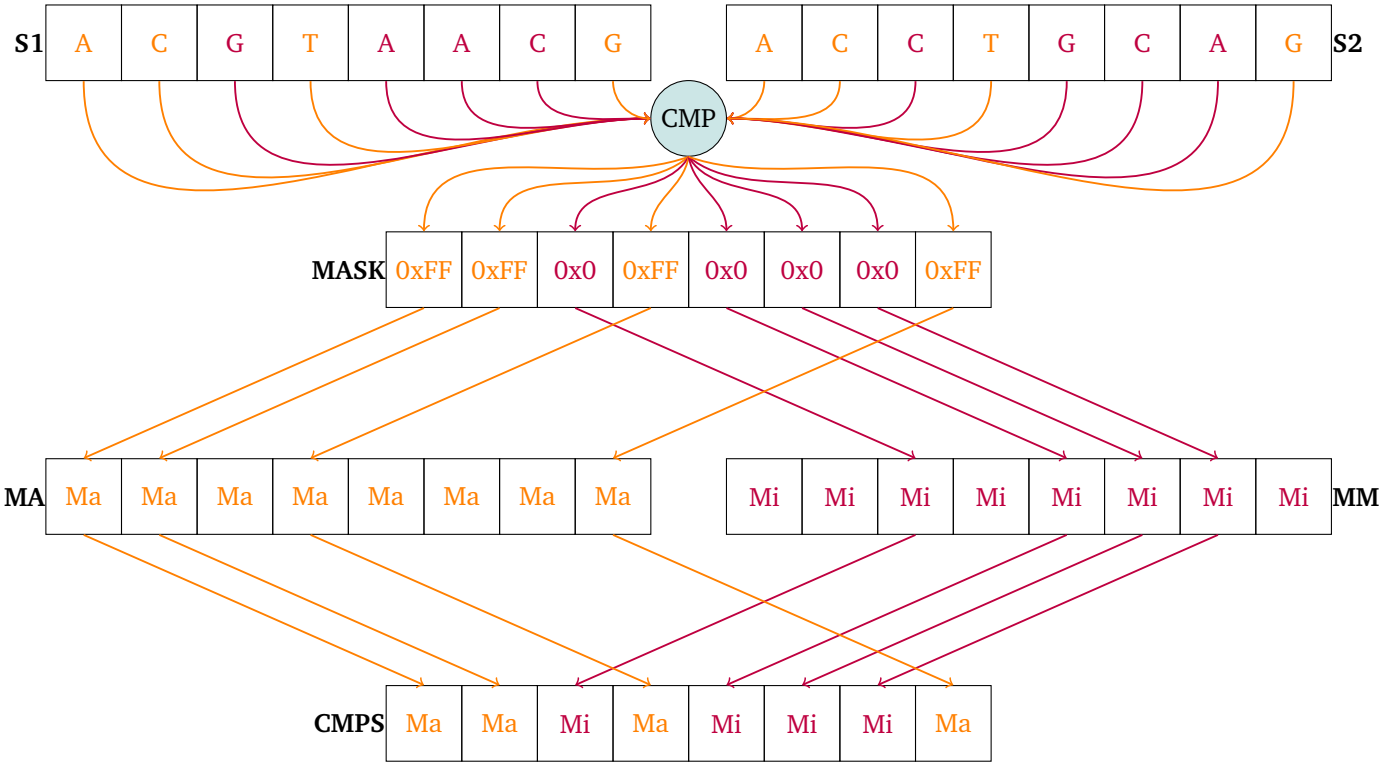


Figure 9: Operaciones realizadas para el calculo de puntajes en la diagonal.

Paso 6: Obtener el máximo entre los valores de la diagonal actual y su indice

En el caso del alineamiento local para obtener eficientemente el máximo de la diagonal realizamos las operaciones ejemplificadas en la figura 10. En cada paso vamos se compara un registro que posee algunos de los valores calculados para la diagonal con otro registro que posee los mismos desplazados a la derecha una cantidad de posiciones. Luego de repetir este paso 3 veces se obtiene el valor máximo entre los 8 words de la diagonal inicial.

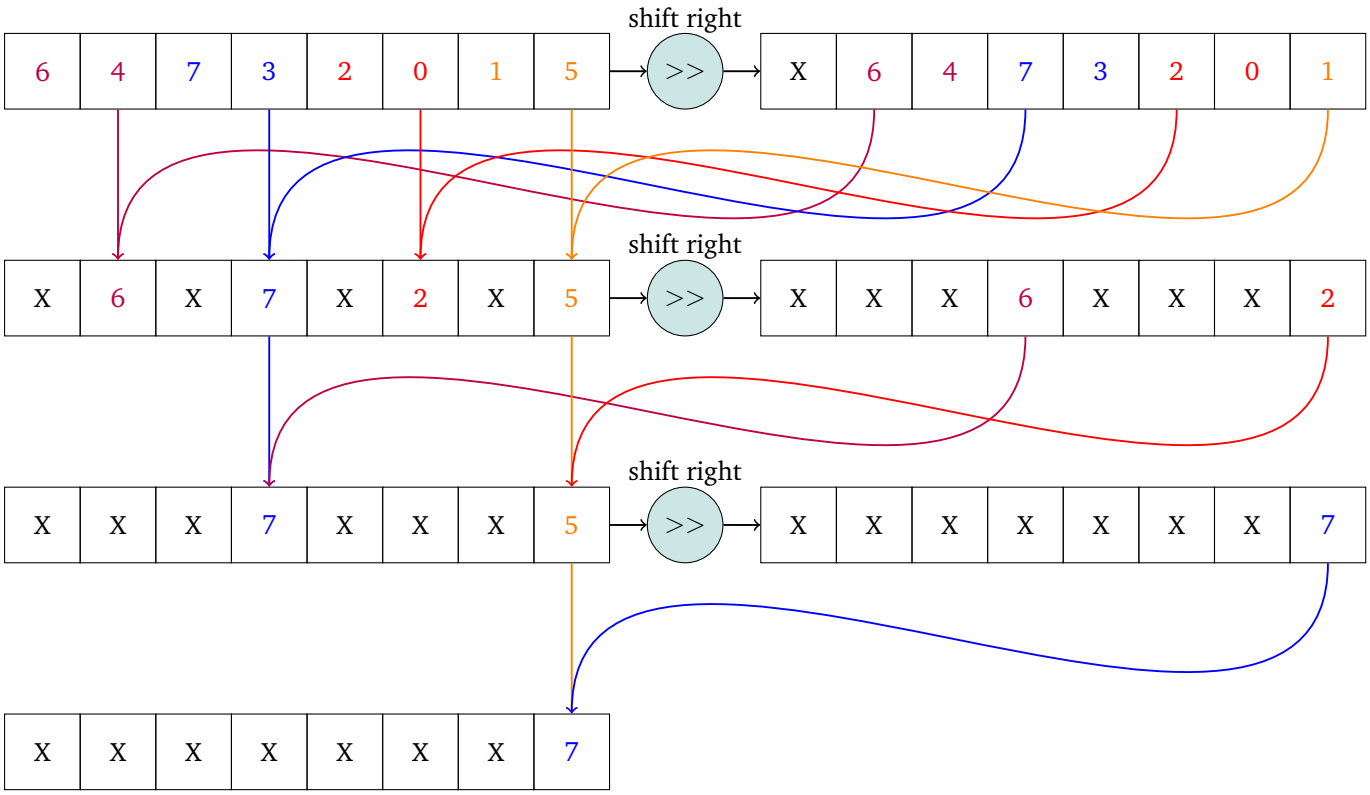


Figure 10: Figura representando las operaciones realizadas para obtener el valor máximo de la diagonal.

Una vez calculado el valor máximo, resta ver la posición del máximo en el registro inicial que representa la diagonal. En la figura 11 se muestran las operaciones realizadas para obtenerlo, realizando tres simples pasos. El primer paso involucra generar una mascara que indique donde se encuentra el o los valores máximos, lo cual puede llevarse a cabo comparando el registro de la diagonal con otro que tenga en todas sus posiciones el valor máximo, el cual se llama *MAX* en la figura 11. Una vez hecho esto, se tiene la máscara **MASK WORDS** la cual va a tener $1's$ en la posición donde está el valor máximo, y como segundo paso se la empaqueta a esta de word a byte, obteniendo el registro **MASK BYTES**. En este punto se tiene una mascara de 8 bytes, donde cada byte vale **0xff** si se corresponde a un máximo, y 0 sino. El paso final es obtener la posición de alguno de estos **0xff**, y esto se puede hacer utilizando la instrucción **BSF** (Bit Scan Forward), devuelve el valor correspondiente al índice del primer bit en 1 que encuentre. Cómo cada 8 bits tenemos un valor diferente, dividimos este índice por 8 y de esta manera obtenemos el índice correspondiente al valor máximo dentro de la diagonal.

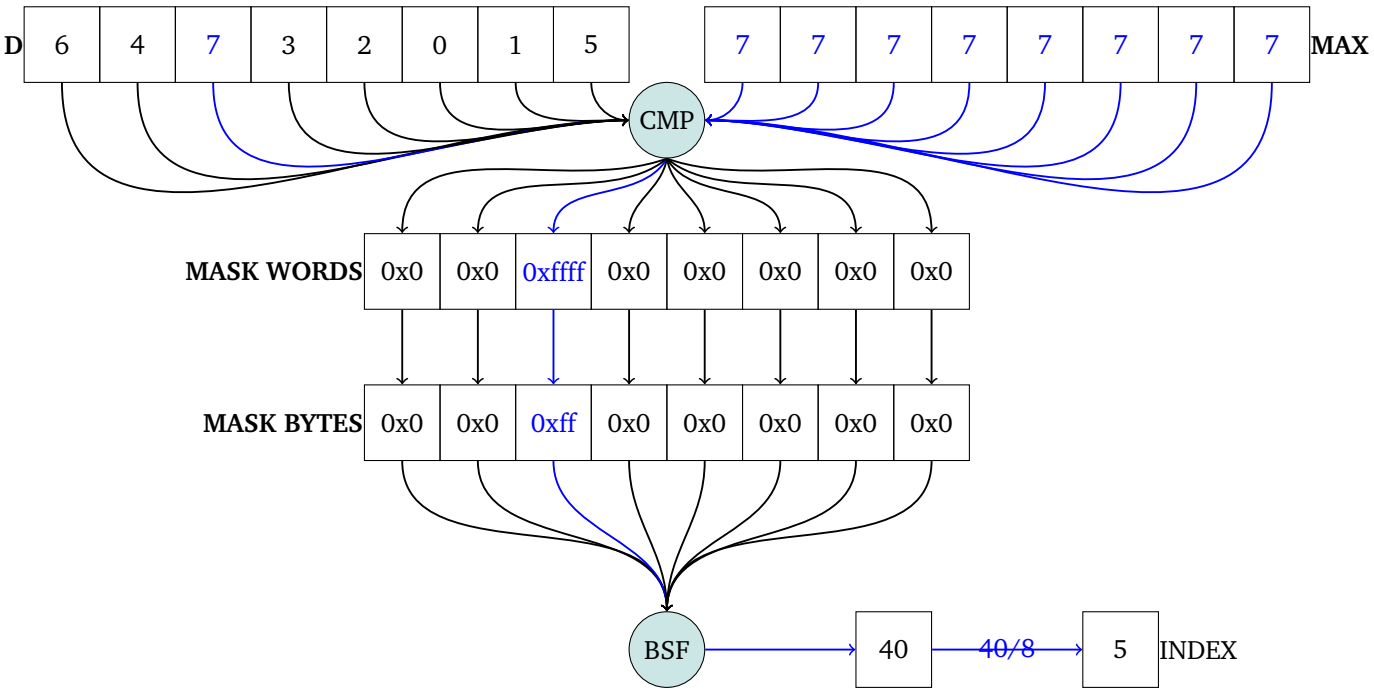


Figure 11: Diagrama representando las operaciones realizadas para obtener el índice del valos máximo en la diagonal

2.3.3. AVX

Al igual que las implementación realizadas con SSE, estas mantienen una estructura y lógica similar en la mayoría de los pasos, solo que trabajando con registros *YMM* en lugar de *XMM*.

Para evitar redundancia en las explicaciones resaltamos específicamente las diferencias sustanciales con respecto a las implementaciones con SSE.

Paso 1 y 2: Modificaciones para realizar los shifts Hay una gran diferencia en como tratamos los casos patológicos de caracteres inválidos al leer las secuencias, tanto de $s1$ como $s2$. Lo que ocurre en estos casos es que la solución con SSE no escala para este caso, ya que ahora se tienen 16 caracteres en un XMM, y no existe una instrucción para shiftear a nivel bytes de manera variable, es decir, solo puede hacerse una cantidad constante.

Una posible forma para solucionar esto a la que llegamos es simular la operación de shift, tanto hacia izquierda o derecha, pero con la opción de poder desplazar una cantidad variable. Para hacer esto utilizamos la instrucción *shuffle*, suministrando una mascara apropiada para *permutar* los valores y así simular la idea de un desplazamiento en alguna dirección. Para visualizar esto primero presentamos un ejemplo a menor escala para 4 caracteres en la figura 2.3.3, de los pasos a llevar a cabo para el caso de la secuencia **s2**, en el cual se quiere hacer un shift a derecha. Se tiene inicialmente el registro *SHIFT MASK*, en el cual cada posición contiene en sus primeros 2 bits el valor de su posición, y los demás bits se encuentran en 1 excepto por el mas alto. La idea de esta mascara es que al sumarle a cada posicion un numero positivo y realizar una permutación con ella, se logra *rotar* los valores del registro hacia la derecha la cantidad de posiciones que se haya sumado. Por ejemplo en la figura 2.3.3 se le suma a *SHIFT MASK* el registro *SHIFT OFFSET*, sumandole el valor 2 a cada posición y obteniendo como resultado el registro *MASK*. Posteriormente se utiliza a este como seleccionador para el shuffle y se permuta la

subsecuencia *TGCA* en el registro *s2*, obteniendo como resultado *CA*, que es efectivamente un shift a derecha de 2 posiciones.

Resta ver ahora como poner valores especiales para que no haya coincidencias psoteriormente, como en el caso de SSE mencionado previamente. Para esto entra en juego los demas bits de la mascara *SHIFT MASK*, los cuales hacen que al producirse un overflow en una posición se ponga en 1 el bit mas significativo, lo cual representa en nuestro caso que esa posición no es un caracter valido. Esto se aprovecha posteriormente a traves de la instrucción blend, ya que se puede seleccionar con ese 1 que las posiciones invalidas pasen a tener un valor conveniente, por ejemplo el mismo de la mascara. Como este valor tiene un 1 en la posicion mas significativa primero es distinto a 0, y segundo ningún carácter del abecedario contiene ese 1 en su valor ASCII, por lo que nunca coincidirá con ninguno.

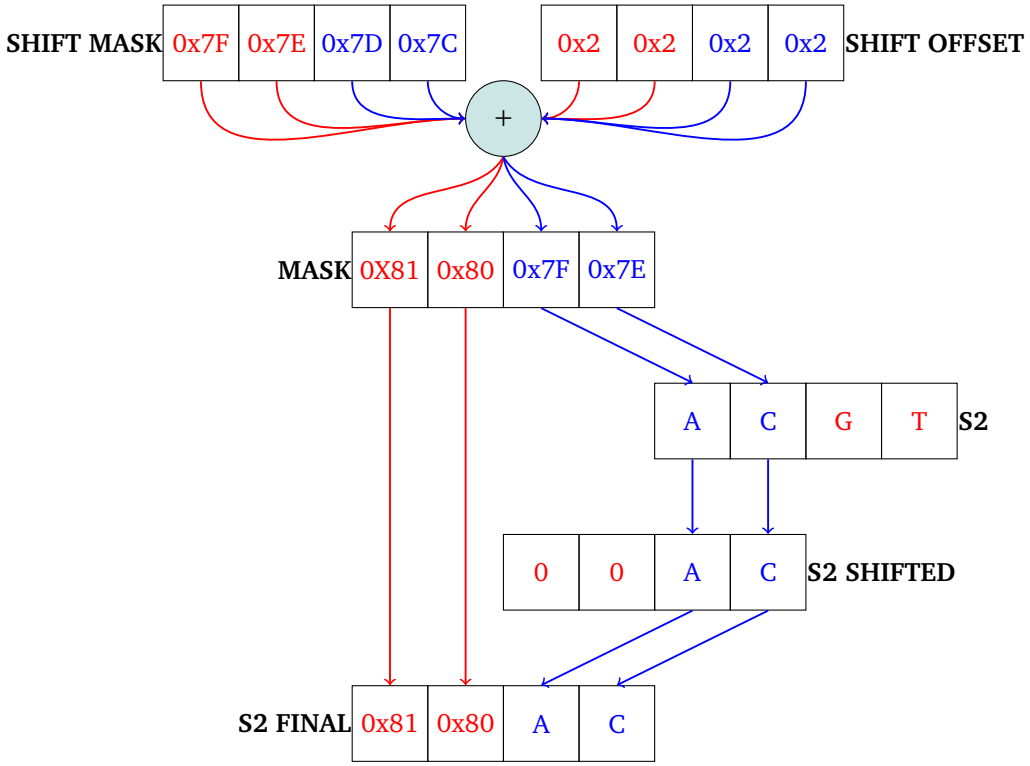


Figure 12: Operaciones realizadas para tratar el caso de tener caracteres inválidos en la parte alta del registro cuando trabajamos con una subsecuencia de **s2**

Para el caso de la secuencia *s1*, tenemos que tratar con dos casos, el desborde por izquierda (primeras diagonales) o por derecha (ultimas diagonales). El procedimiento para el desborde por derecha es similar al explicado previamente, solo que se omite el paso del blend, ya que ya se generan 0's en las posiciones adecuadas como se puede ver en la figura 2.3.3.

Luego en la figura 2.3.3 se observa como se trata el caso de desborde por derecha para una subsecuencia de **S2**. Con una lógica similar a la explicada por la figura 2.3.3 se utiliza la resta entre **SHIFT MASK**, la cual tiene directamente el valor asociado a su posición esta vez, y **SHIFT OFFSET** para generar el registro **MASK** de manera que una vez más tenga el bit mas significativo en 1 para caracteres invalidos y en los restantes el indice del caracter que se desplaza alli. Luego se realiza de manera analoga un shuffle de a bytes entre **S1** y **MASK** generando **S1 SHIFTED**, donde se encuentran los caracteres desplazados la cantidad de lugares que indican los valores de **SHIFT OFFSET** y dejando en 0 los lugares donde había caracteres inválidos, igual que en el caso de desborde por derecha.

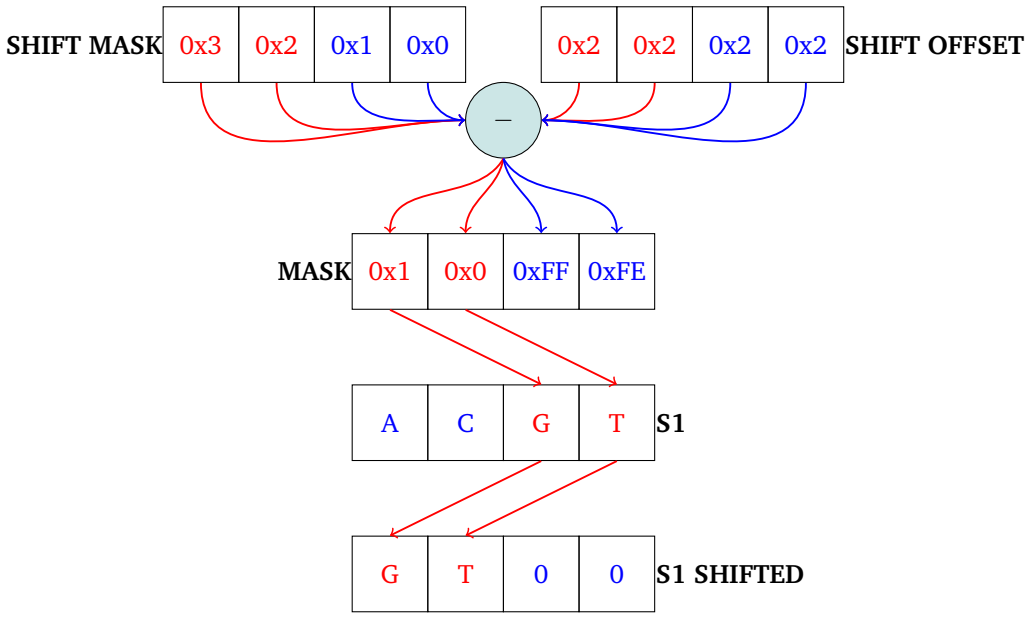


Figure 13: Operaciones realizadas para tratar el caso de tener caracteres inválidos en la parte baja del registro cuando trabajamos con una subsecuencia de **s1**

Paso 1 & 2: Desempaquetado de los caracteres (bytes) a words Al intentar desempaquetar los valores de los caracteres a procesar para trabajar a nivel word nos encontramos con que debemos desempaquetar la parte alta por un lado y luego la parte baja por el otro para luego combinar ambas partes en un solo registro YMM. Esto se debe a que la instrucción disponible para hacerlo solo desempaqueta los 8 bytes mas altos o mas bajos de un XMM.

Paso 3: Obtener los scores provenientes de las direcciones arriba y diagonal En el cálculo de los puntajes de las direcciones de arriba y diagonal debemos realizar un desplazamiento una vez que se leen los valores de la diagonal correspondiente en cada caso. El problema con el que nos encontramos es que la instrucción para realizar shifteos a nivel byte para un registro *YMM* disponible lo hace individualmente por cada linea de 128 bits del mismo (los 128 bits mas altos y los 128 bits mas bajos del registro YMM). Como queremos shiftear a derecha 2 bytes, necesitamos que el word en la posición mas baja de la linea de 128 bits alta se mueva a la posicion mas alta de la linea baja de 128 bits. Por lo tanto, para solucionar esto, optamos por extraer este word que se va a perder al shiftear e insertarlo luego de realizar el shift en la posición más alta de la linea baja de 128 bits. De esta manera mantenemos la lógica realizada para las implementaciones de SSE en esta parte

Paso 6: Obtener la posición máxima

Finalmente en el alineamiento local debemos realizar un paso extra en la actualización de la posición máxima. Esto se muestra en la figura 14 donde se requiere un segundo registro y una comparación extra para obtener el valor máximo en la diagonal.

Si se repite exactamente el mismo procedimiento explicado para SSE para obtener el valor máximo, pero para registros *YMM*, obtenemos como resultado un registro que tiene un valor máximo para cada parte de 128 bits del registro *YMM*. Una vez más no podemos hacer un paso mas de lo explicado previamente para SSE, ya que por lo que se mencionó anteriormente sobre la instrucción shift para *YMM*, no podemos pasar facilmente un word de la parte alta de 128 bits a la parte baja del *YMM*. Haremos uso una vez mas de la instrucción shuffle, pero a nivel 8 bytes y para *YMM*, llamada en este

caso **vpermq**, para poder mover el valor deseado a un lugar conveniente. Para lograr esto hacemos uso de **vpermq** como se muestra en la figura 14, donde la mascara **MASK** tiene 8 bits y se usa como seleccionador, y cada letra dentro de los registros **NUMS** y **NUMS AUX** representa un word. El shuffle que se lleva a cabo copia el quadword en la parte baja de la parte alta de 128 bits de **NUMS**, el cual contiene uno de los máximos posibles, en el quadword de la parte baja de la parte baja de 128 bits del **NUMS AUX**, como muestra la figura 14.

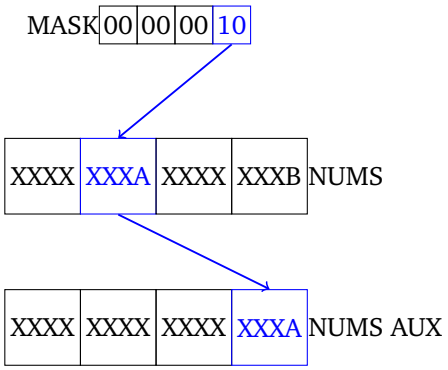


Figure 14: Paso extra en comparación con los realizados en las implementaciones SSE para obtener el valor máximo de la diagonal.

2.3.4. AVX-512

Estas implementaciones mantienen de manera similar la lógica de la paralelización con respecto a las de SSE y AVX, con unas notables diferencias que vale la pena remarcar. Recordemos primero que esta extensión de instrucciones nos permite trabajar con los registros *ZMM* de 512 bits, lo cual a su vez nos permite procesar simultáneamente de a 32 caracteres. A su vez, esta extensión nos proporciona con un nuevo tipo de registros y nuevas funcionalidades para las instrucciones utilizadas hasta el momento, permitiendo así nuevos enfoques a la hora de realizar los pasos de la paralelización.

Paso 1 y 2: Obtener las secuencias **s1** y **s2** de memoria Tanto para las implementaciones de SSE como de AVX los problemas que surgían se solucionan leyendo desfasado de memoria con un cierto offset y aplicando un shift sobre los caracteres para reubicarlos correctamente. Si bien en este caso se puede buscar hacer un shift nuevamente, decidimos atacar la raíz del problema, el cual era hacer lecturas fuera de la memoria y generar así una violación de segmento. La solución a la que arribamos en este caso fue hacer uso de la instrucción **vmovdqu8**, que permite leer de memoria 16, 32 o 64 bytes y cargarlos en un registro apropiado, y que tiene dos parámetros opcionales útiles **{k}** y **{z}**. Utilizando el parámetro **{k}** podemos aplicar una máscara de bits que permite seleccionar los bytes que queremos leer de memoria o no, a partir de cierto base address. Si el bit correspondiente a una posición es 1, entonces lee ese byte y lo pone en el lugar correspondiente en el registro destino. En caso contraria, la instrucción puede utilizarse en dos modos, dependiendo de si se especifica o no **{z}**, para mantener el valor que se encuentra en el registro destino o bien para ignorarlo y colocar un cero. La parte mas importante en todo este razonamiento es que, en el caso de que el bit de la mascara **k** se encuentre en 0 y la posición de memoria correspondiente sobre la cual no se realizo una lectura era invalida, no se produce una violación de segmento.

Mostramos a continuación en la figura 15 un ejemplo a escala del uso de **vmovdqu8**, trabajando con registros de 8 bytes en lugar de 32, para el caso de leer la subsecuencia de **s2** cuando no hay suficientes

caracteres validos. En este caso **MEM** refleja la porción de memoria a partir de donde vamos a realizar las lecturas, donde la A de color naranja se encuentra en la posición base address, y las X representan un valor fuera de la memoria valida de **s2**. Primero se genera la mascara **SHIFTED MASK** shifteando la mascara **MASK** a derecha la cantidad de posiciones que serían accesos fuera de memoria De esta manera **SHIFTED MASK** tiene ceros en las posiciones que representan accesos fuera de memoria y 1's en las que no, es decir los caracteres útiles. En este caso. al no usar el parámetro {**z**}, las posiciones que no se leen de memoria se mantienen en el registro destino, en este caso **ONES**, y finalmente el resultado de esta operación se coloca en las posiciones correspondientes en el registro **COL**. Notar que el uso del registro **ONES** presenta una ventaja, ya que al finalizar la instrucción **COL** ya posee valores distintos a 0 y a caracteres del abecedario en las posiciones invalidas, resultando en un uso muy conveniente del mismo. Finalmente podemos ver que el resultado en **COL** es efectivamente la subsecuencia deseada para el ejemplo en la figura 15, en la ubicación correcta.

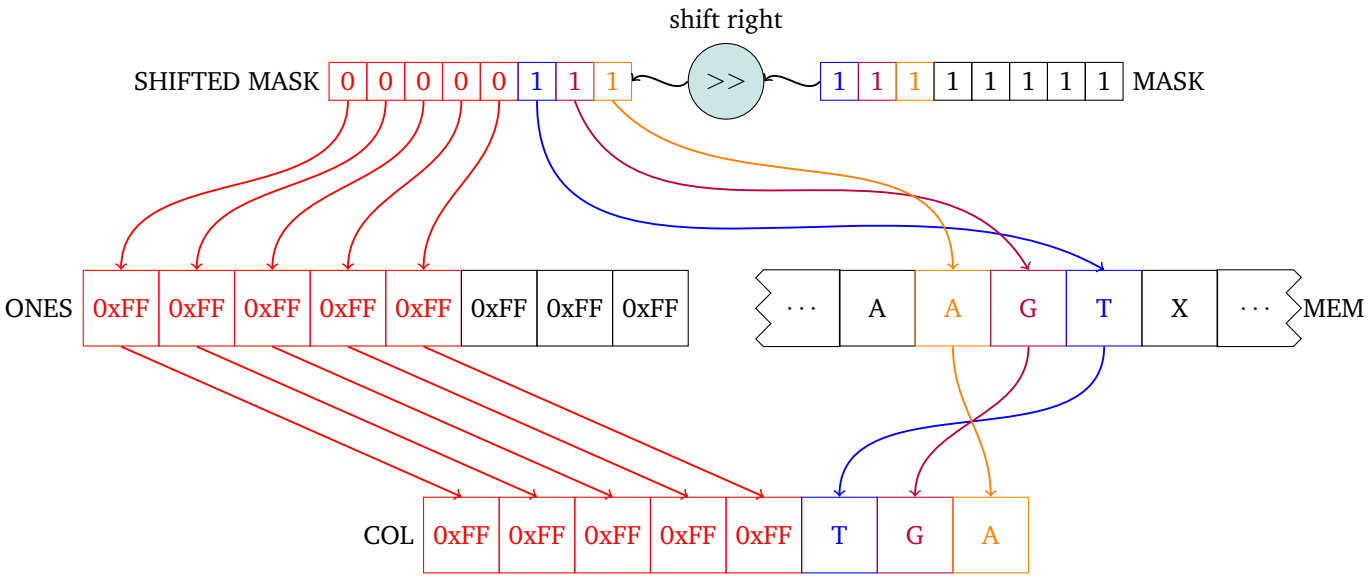


Figure 15: Operaciones realizadas para tratar con el caso donde la subsecuencia de **s2** es menor a lo que vamos a levantar

Análogamente vemos en la figura 16 el caso de desborde por izquierda, como en AVX, para la secuencia **s2**. La única diferencia es que se especifico el parametro **{z}**, para poner 0's en las posiciones asociadas a caracteres inválidos, siguiendo la convención definida desde las implementaciones SSE.

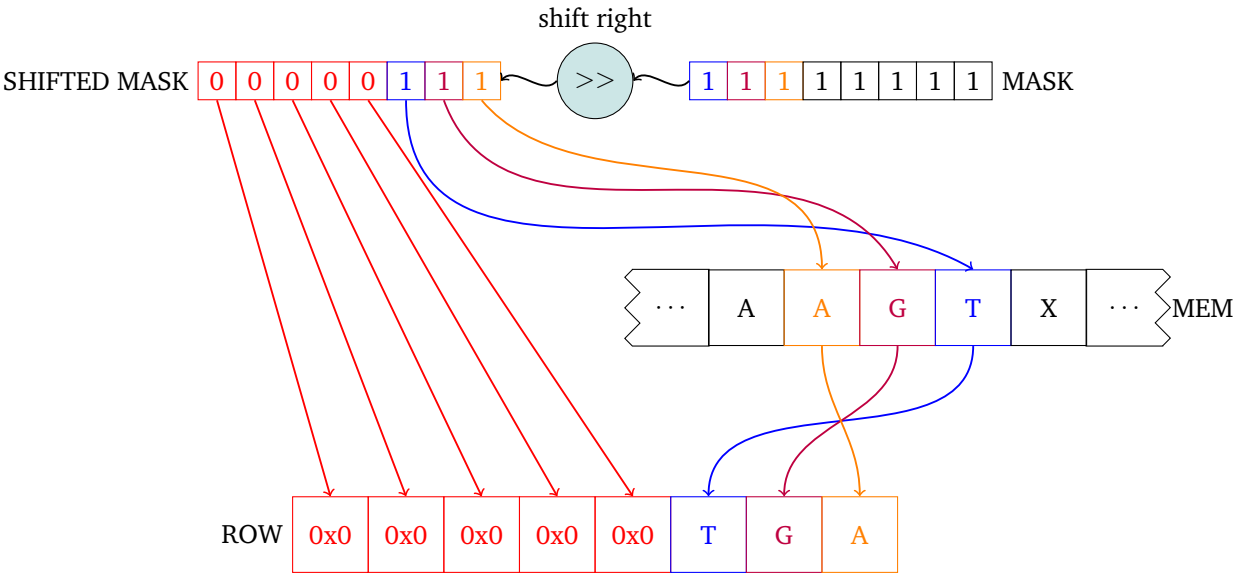


Figure 16: Operaciones a realizar cuando los caracteres que necesitamos de la subsecuencia de **s1** son menos de los que podemos levantar

Paso 1 y 2: Desempaquetado de las subsecuencias **s1** y **s2** A diferencia de AVX, donde se puede desempaquetar los 8 bytes mas altos o bajos del *XMM*, la instrucción equivalente para *ZMM* desempaqueta individualmente los 8 bytes de cada linea de 128 bits. Esto es un problema dada la disposición de los 32 caracteres en el registro *ZMM*, que se encuentran en la parte *YMM* del mismo, ya que el desempaquetado no se haría correctamente. Sin embargo existe una solución eficiente a este problema, que involucra hacer un reordenamiento de los bloques de 8 bytes, previo al desempaquetado con esa instrucción. La figura 17 es una vez mas un ejemplo a escala de esta solución, considerando a cada linea de 32 bits, y a cada bloque de 16 bits.

Utilizamos la instrucción **vpermq** que ya se utilizó en el paso 6 de AVX, en conjunto con la máscara **PERM MASK** para reubicar los caracteres de **S** en **S PERMUTED**, como muestra la figura 17. Luego, a partir de **S PERMUTED** ahora si se puede realizar el unpack entre el registro **S PERMUTED** y **ZEROES**, dejando en **S UNPACK** los valores desempaquetados correctamente para trabajar a nivel word mas adelante con los caracteres de **S**.

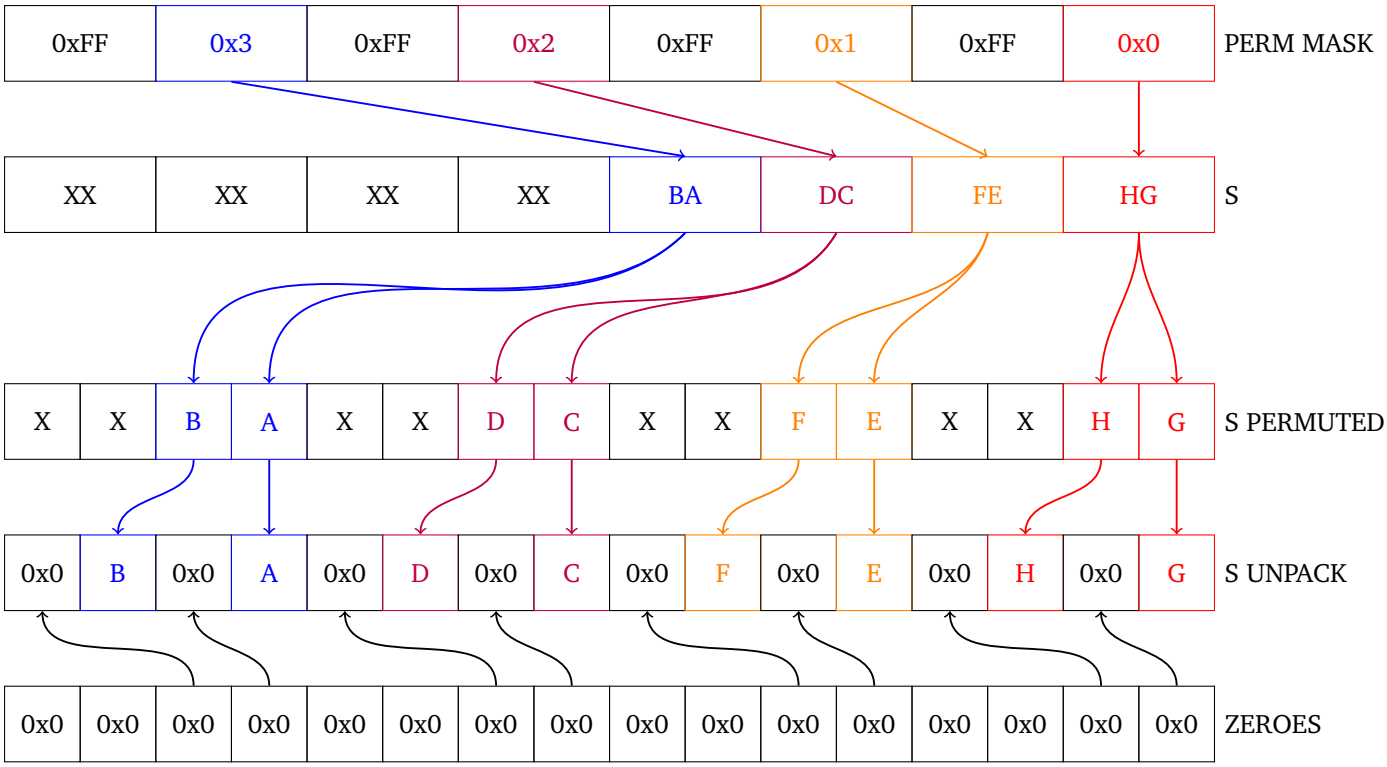


Figure 17: Operaciones a realizar al desempaquetar los caracteres en el registro para trabajar a nivel word

Paso 3: Obtener los scores provenientes de las direcciones arriba y diagonal Al igual que en AVX, donde el problema de este paso se debe a que el shift es individual por cada linea de 128 bits, en *AVX-512* ocurre exactamente lo mismo, pero ahora con 4 lineas de 128 bits. Se puede resolver este problema de manera similar que en *AVX*, extrayendo valores e insertándolos, pero esto puede llegar a ser ineficiente ya que hay 3 valores que requieren este trabajo. Por estos motivos cambiamos el enfoque y utilizamos la instrucción **vpermw** para generar un desplazamiento a derecha una posición a nivel word, que es lo que queríamos. Sin embargo después de hacer esto, es necesario en ambos casos insertar el valor correspondiente del vector auxiliar en la posición mas alta de cada registro desplazado. Esto también es problemático, ya que para hacerlo se debe extraer la linea de 128 bits mas alta del *ZMM*, insertar el valor, y finalmente volver a insertar la linea de 128 bits en la parte mas alta. En lugar de hacer esto, cambiamos lo que antes era un desplazamiento a derecha para que ahora sea una rotación. De esta manera al valor en la parte mas baja, que igualmente iba a ser descartado luego del desplazamiento, se lo reemplaza por el valor del vector auxiliar, y al realizar la rotación se obtiene el resultado de combinar ambos pasos en uno.

Un ejemplo de todo esto se encuentra a continuación, considerando en este caso que se quiere rotar un *XMM* sin afectar a la idea general. Se tiene primero en el registro **SCORES** los valores de la diagonal cargados de memoria, y luego se le inserta a este el valor del vector auxiliar en la parte mas baja. Cabe destacar que esta inserción puede hacerse en este caso con una sola instrucción a diferencia del otro caso, resultando en una opción mucho mas eficiente. Luego utilizando la mascara **ROT MASK** con los valores adecuados junto con el registro **SCORES** y la instrucción para hacer la permutación se lleva a cabo la rotación de un word a derecha, obteniendo así **FINAL SCORES**, que es exactamente lo que se buscaba.

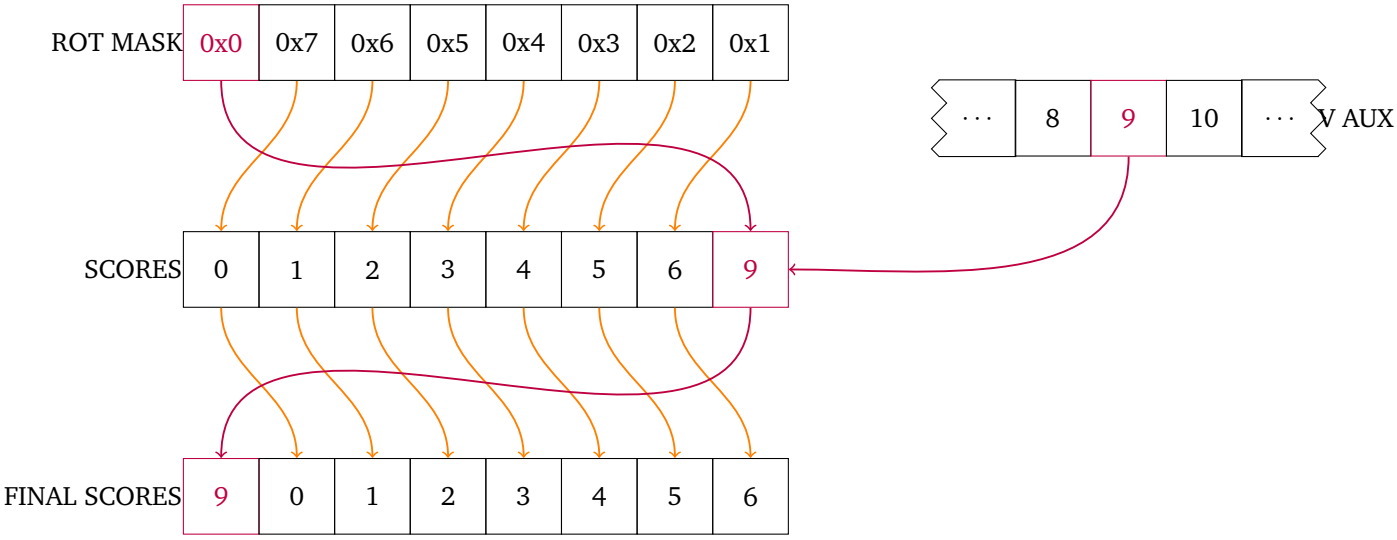


Figure 18: Operaciones a realizar cuando queremos shiftear los registros para calcular los puntajes de las direcciones de arriba y diagonal

Paso 6: Obtener la posición máxima De manera análoga que en las implementaciones anteriores con *AVX* para actualizar la posición máxima, donde teníamos en un determinado momento un valor máximo por cada linea de 128 bits, en *AVX-512* sucede lo mismo, pero ahora con 4 máximos en lugar de 2. El objetivo una vez más es reubicar los 4 máximos en una sola linea de 128 bits, para así poder seguir con los mismos pasos de shift y tomar máximo que se mencionó en las implementaciones con *SSE*. Sin embargo, en lugar de repetir la idea que utilizó en *AVX* haciendo uso de las permutaciones, decidimos utilizar un nuevo tipo de instrucción que aparece en *AVX-512*, que se llama compress. Como el nombre indica, esta instrucción recibe un registro, una mascara, y selecciona a partir de esta última los valores del registro que quieren conservarse, para así después guardarlos en otro registro, pero de manera contigua. Esta instrucción como se puede ver no conserva las posiciones originales, pero si el orden de aparición entre los valores seleccionados, generando asi la idea de una *compresión* de los datos.

A continuación mostramos el caso de uso que le dimos a esta instrucción en la figura 19, donde cada carácter de **DIAG** y **PERM DIAG** representa un word. La versión de la instrucción que se usa en la figura es **vpcompressd**, que trabaja a nivel doubleword, y **BITMASK** es el parámetro {**k**}, que es la máscara que permite seleccionar los valores como se explicó previamente. **BITMASK** selecciona los 32 bits mas bajos de cada linea de 128 Luego, obteniendo asi dentro de cada uno de estos 32 bits el máximo word de cada linea. Finalmente, en el registro **PERM DIAG** se visualiza el resultado de la compresión, donde se tienen los 4 máximos en una sola linea, listos para realizar dos comparaciones de a pares para luego quedarnos con el valor máximo entre los 32 words.

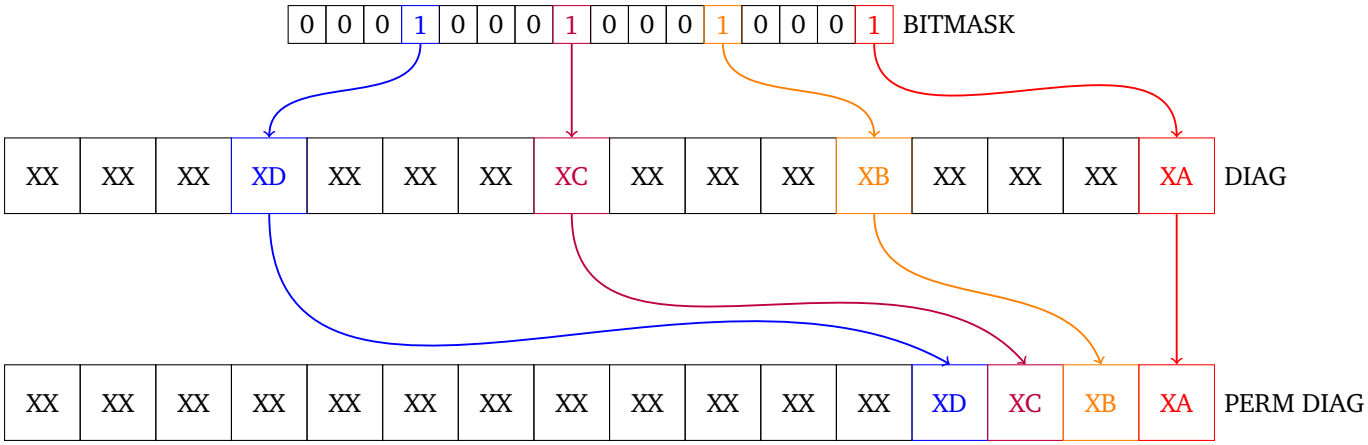


Figure 19: Operación extra requerida para obtener el valor máximo de la diagonal

2.4. Experimentos

Al momento de compilar las distintas versiones hechas en C para la experimentación, se forzó al compilador para que utilice instrucciones permitidas para cada versión particular. Por ejemplo, en el caso de las versiones de SSE, se compiló para que utilicen solo SSE, y análogamente en el de AVX para utilizar SSE y AVX. El objetivo de esto es que simule un caso de uso real, donde se utiliza la versión mas alta que soporte el CPU disponible. En los experimentos que involucran instancias con secuencias aleatorias, las generamos seleccionando del abecedario $[A, G, C, T]$ al azar la cantidad de veces necesaria para construir la secuencia del largo buscado.

A continuación se detallan los experimentos realizados junto con las hipótesis que los acompañan.

2.4.1. Secuencias aleatorias

> ***Hipótesis 1.*** *Para este experimento nuestra hipótesis es que, a la hora de comparar los tiempos de dos implementaciones para un mismo algoritmo, siempre es superior el que usa una mejor tecnología. Pero, en el caso de que esta coincida, la que usa lenguaje ASM es superior. Suponemos esto último debido a que no hay optimizaciones presentes en las versiones que utilizan C, por lo que resulta difícil que al tener una capa de abstracción como C se logre una mejor performance que ASM.*

Queremos observar como se comportan las distintas implementaciones para pares de secuencias totalmente aleatorias, pero que coincidan en longitud. Para lograr esto proponemos un experimento donde corremos las distintas versiones, es decir, variando el lenguaje entre C y ASM, y la tecnología utilizada entre LIN, SSE, AVX y AVX512. Con esto vamos a tener 7 combinaciones posibles, ya que no existe la variante en lenguaje ASM y tecnología LIN. Cada una de estas a su vez sera ejecutada para cada tipo de algoritmo, Needleman-Wunsch y Smith-Waterman. Es importante aclarar que todas estas versiones fueron compiladas sin flag de optimización, por lo que ninguna versión en C se encuentra beneficiada en la compilación. Dada esta configuración, buscamos observar como se ven afectados los tiempos de

las distintas versiones a medida que aumenta la longitud de ambas secuencias, y poder ver también la relación que hay entre las implementaciones que coinciden en tecnología pero difieren en lenguaje.

2.4.2. Secuencias de genomas virales

> ***Hipótesis 2.*** *En este experimento comparamos el desempeño de las distintas implementaciones del algoritmo Needlman-Wunsch alineando genomas virales. Nuestra hipótesis es que se mantendrán las mismas relaciones observadas en el experimento de secuencias aleatorias debido a que el desempeño debería ser independiente del contenido de cada secuencia para cada longitud evaluada.*

Para realizar este experimento se obtuvieron secuencias de genomas virales tomadas del repositorio NCBI Virus. Estas secuencias se distribuyen en formato FASTA, por lo tanto se implementó un *parser*[3] para poder leer archivos de este tipo. Se seleccionaron genomas de distinto tamaño para cubrir todo el rango en el cual las implementaciones del algoritmo pueden trabajar. Para generar secuencias menores a 5000 pb se utilizaron distintos segmentos del virus de la gripe H1N1. Adicionalmente se emplearon secuencias de HPV, HIV, Zika, Ebola, Marburg, SARS-CoV 2 y MERS. Para cada genoma viral o segmento, se realizó el alineamiento al genoma de referencia de 5 secuencias correspondientes al mismo genoma pero distintas al de referencia. Al igual que en el experimento anterior, se utilizaron las 7 combinaciones de lenguaje y tecnología previamente mencionadas y las implementaciones fueron compiladas sin flag de optimización. El objetivo de este experimento es verificar el desempeño observado previamente para las distintas implementaciones del algortimo Needleman-Wunsch pero con casos de uso reales.

2.4.3. Simulación de lecturas NGS

> ***Hipótesis 3.*** *En este experimento comparamos el desempeño de las distintas implementaciones del algoritmo Smith-Waterman alineando secuencias simuladas para ser similares a las lecturas generadas por los secuenciadores de nueva generación (NGS). Nuestra hipótesis es que se mantendrán las mismas relaciones observadas en el experimento de secuencias aleatorias debido a que el desempeño debería ser independiente del contenido de cada secuencia para cada longitud evaluada.*

Para realizar este experimento se simularon las lecturas de NGS a partir de los genomas virales de referencia empleados en el experimento anterior utilizando el programa Mason. Se generaron 10 secuencias de tipo Illumina de 100 pb por genoma de referencia y se alinearon a dicho genoma. Al igual que en el experimento anterior, se utilizaron las 7 combinaciones de lenguaje y tecnología previamente mencionadas y las implementaciones fueron compiladas sin flag de optimización. El objetivo de este experimento es verificar el desempeño observado previamente para las distintas implementaciones del algortimo Smith-Waterman pero con casos de uso reales.

[3] En los archivos *FASTA.cpp/hpp*

2.4.4. Secuencias aleatorias O3

En el experimento 1 se emplean secuencias aleatorias y se compilan las implementaciones hechas en lenguaje C sin optimizaciones para que no se vean beneficiadas por estas. Nos parece interesante ver de manera similar el caso opuesto: como puede mejorar el desempeño de las implementaciones en lenguaje C permitiendo optimizaciones. Por esto definimos este nuevo experimento de manera análoga al primero, pero ahora utilizando el flag de optimización **O3** para compilar para todas las versiones en dicho lenguaje. El objetivo en este caso es ver los cambios en cuanto a tiempos con respecto al otro experimento y si alguna implementación que no es superior a otra sin optimizaciones lo es al aplicarlas. Esto puede resultar especialmente útil si al hacer uso de estas optimizaciones una versión en C supera a su respectiva en ASM, porque implicaría que no es necesario llegar a bajo nivel para lograr una buena performance.

> ***Hipótesis 4.*** *En este caso la hipótesis resulta similar a la del experimento 1. Suponemos al igual que antes que, al comparar dos versiones con distintas tecnologías, es superior la de mejor tecnología. Pero, en este caso, con la diferencia de que si se trata de la misma tecnología, la versión C resulta superior. El cambio con respecto a la otra hipótesis sobre que lenguaje es superior es debido a que las optimizaciones pueden llegar a equilibrar la balanza y beneficiar más a la versión en C.*

2.5. Ejecución en AWS

Amazon Web Services (AWS) es una subsidiaria de Amazon que ofrece una API y plataformas de computación en la nube según la demanda requerida por individuos, empresas y gobiernos, con un sistema de pago por uso medido. Estos servicios web de computación en la nube proporcionan una variedad de infraestructura, componentes y herramientas de sistemas distribuidos. Uno de estos servicios es Amazon Elastic Compute Cloud (EC2), que permite a los usuarios tener a su disposición un cluster virtual de computadoras, disponible todo el tiempo, a través de Internet. Este servicio de AWS virtualiza la mayoría de los atributos de una computadora real, incluidas las unidades de procesamiento central y las unidades de procesamiento de gráficos, memoria RAM, almacenamiento en disco duro o SSD, distintos sistemas operativos, redes y aplicaciones precargadas, como servidores web, bases de datos y gestión de relaciones con el cliente (CRM). Dado que ninguno de los integrantes de este trabajo posee acceso a un procesador con el set de instrucciones AVX-512, se utilizó el servicio EC2 para disponer de una instancia M5 que permitiera realizar la experimentación en estas condiciones.

3. Experimentación

Para realizar la experimentación sobre las diversas implementaciones es necesario contar con un procesador cuya arquitectura sea compatible con todas las instrucciones utilizadas en ellas. En nuestro caso, todas las implementaciones fueron hechas para mantener compatibilidad con la arquitectura skylake, siendo esta la utilizada para todas las pruebas. Dentro de los posibles CPU que poseen esta arquitectura decidimos utilizar el que es versión servidor, **skylake-avx512**, el cual soporta todas las instrucciones AVX-512 presentes en las implementaciones. Debido a que no contamos con este CPU llevamos a cabo la experimentación en una instancia de AWS que si contara con este, siendo estas las especificaciones de la misma:

Primer instancia :

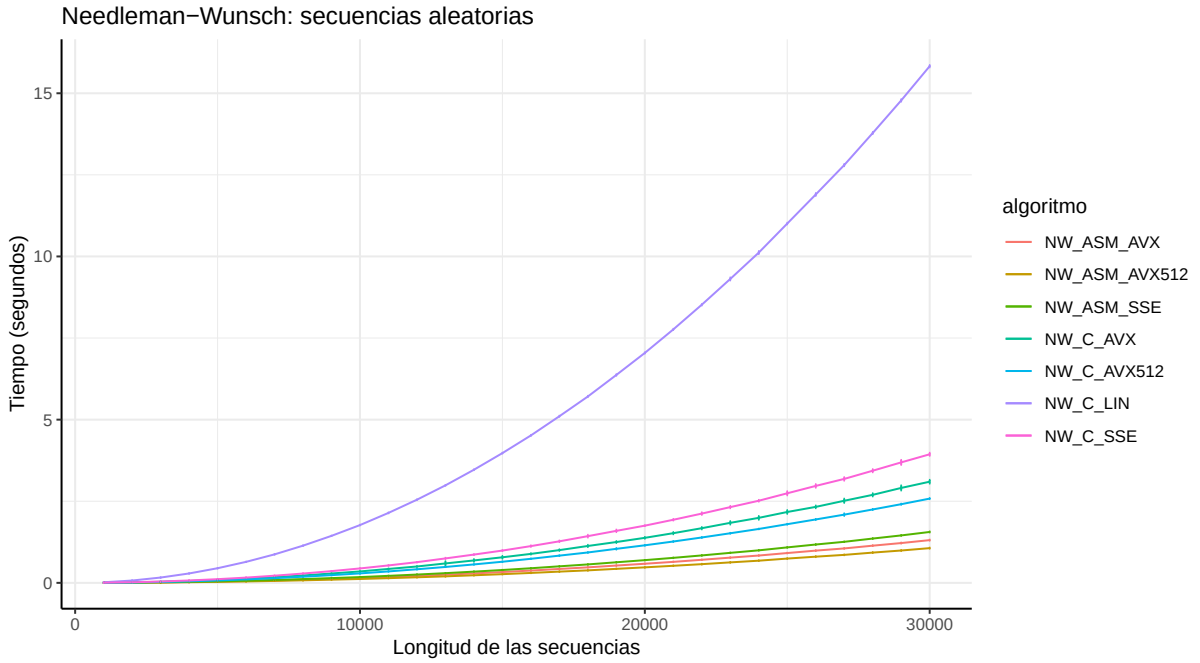
```
Architecture:
CPU(s):                          2
Model name:                      Intel(R) Xeon(R) Platinum 8124M CPU @ 3.00GHz
CPU MHz:                         3304.867
L1d cache:                       32 KiB
L1i cache:                       32 KiB
L2 cache:                        1 MiB
L3 cache:                        24.8 MiB
```

Segunda instancia :

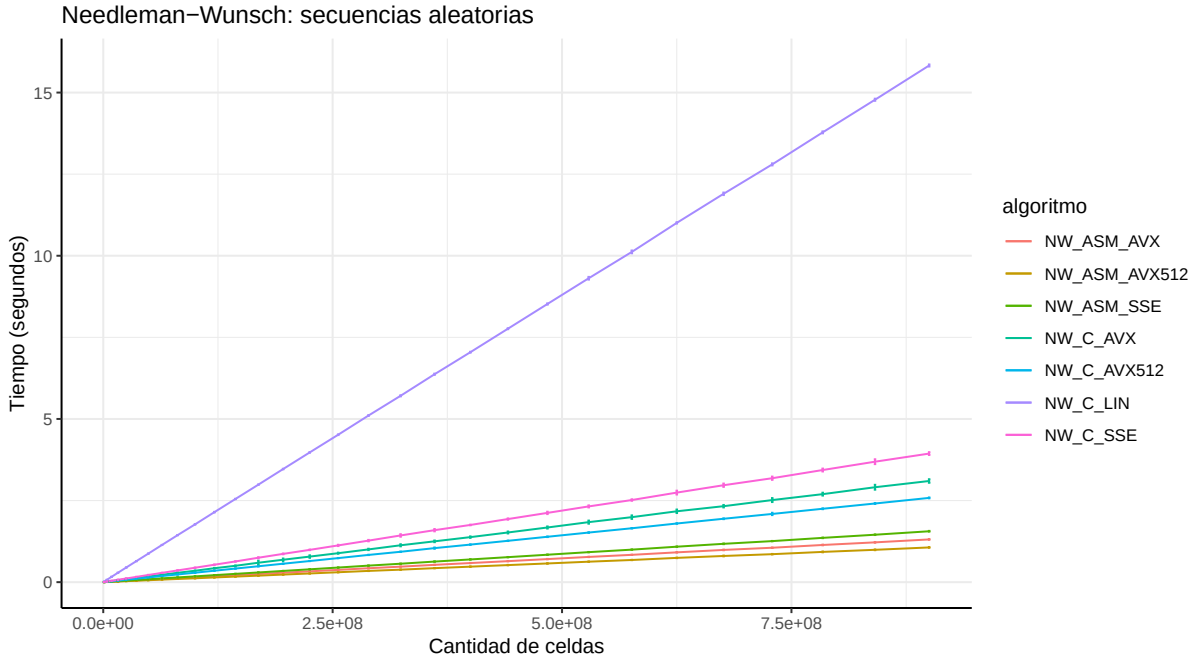
```
Architecture:
CPU(s):                          2
Model name:                      Intel(R) Xeon(R) Platinum 8275CL CPU @ 3.00GHz
CPU MHz:                         3642.837
L1d cache:                       32 KiB
L1i cache:                       32 KiB
L2 cache:                        1 MiB
L3 cache:                        35.8 MiB
```

3.1. Secuencias aleatorias

A continuación presentamos los resultados de este experimento, primero para las versiones del algoritmo Needleman-Wunsch y luego para Smith-Waterman. En la figura 20 observamos, en el gráfico superior, los tiempos para las distintas implementaciones en función de las longitudes de las secuencias y, en el gráfico inferior, en función de la cantidad de celdas presentes en la matriz de programación dinámica:



(a) Tiempos de ejecución para las distintas implementaciones del algoritmo Needleman-Wunsch en función de las longitudes de las secuencias.



(b) Tiempos de ejecución para las distintas implementaciones del algoritmo Needleman-Wunsch en función de la cantidad de celdas en la matriz de programación dinámica.

Figura 20

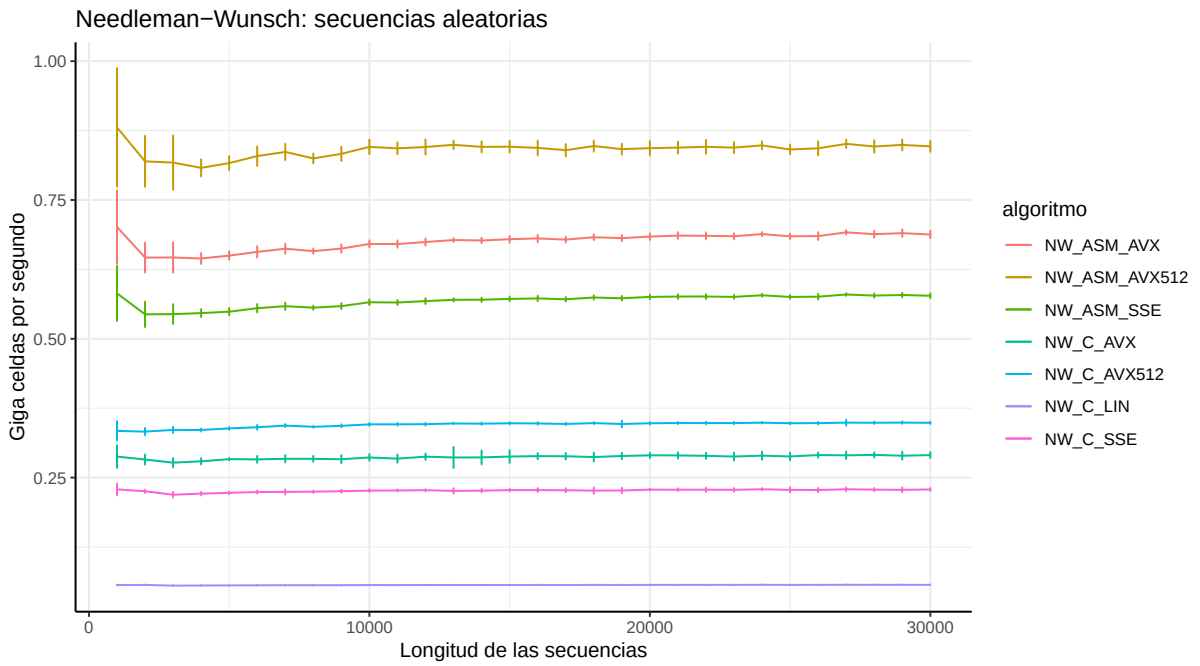
Lo primero que se aprecia en la figura 20a es el comportamiento cuadrático en función de la longitud de las secuencias y, en la figura 20b, el comportamiento lineal cuando aumenta la cantidad de celdas. Esto se relaciona directamente con la complejidad del algoritmo de $O(nm)$, donde n y m son el largo de las secuencias. Se puede pensar que, en este caso, al ser $n = m$ la complejidad es $O(n^2)$. Si se analiza como aumenta el tiempo de ejecución cuando aumenta el n, se observa claramente el comportamiento cuadrático. Para el otro caso, como la cantidad de celdas es n^2, la complejidad se comporta lineal con

respecto a este numero y, por lo tanto, es de esperar un crecimiento lineal del tiempo con respecto a la cantidad de celdas como se observa en la figura.

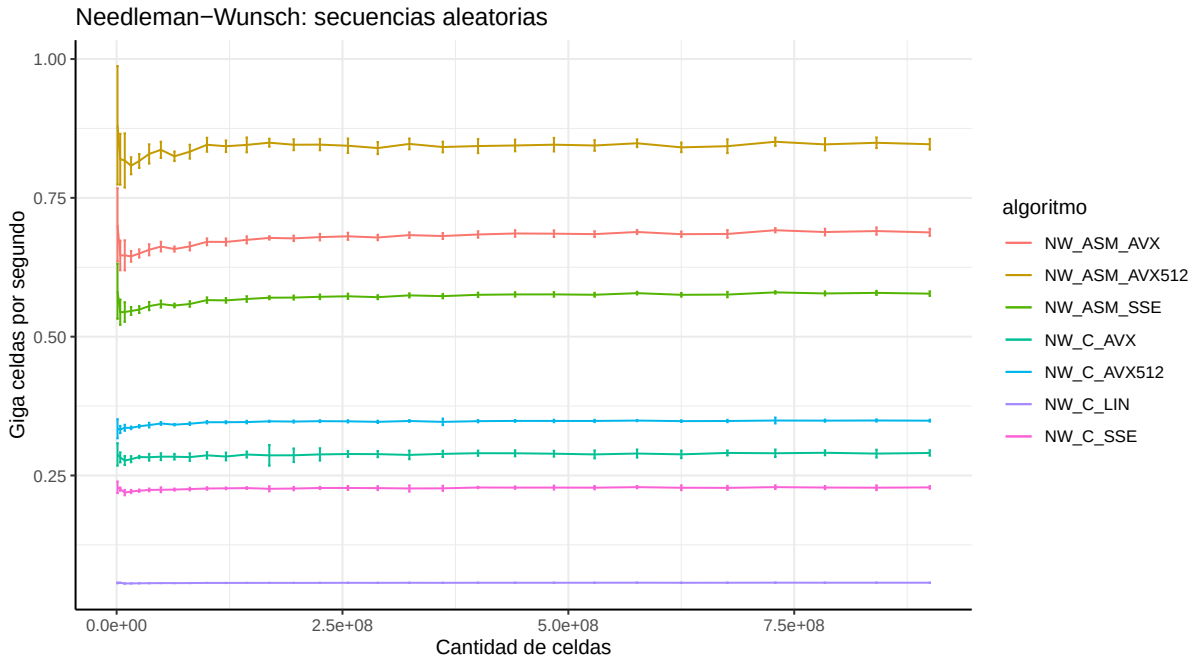
Si nos concentramos ahora en las relaciones de tiempos entre las implementaciones, en ambas figuras hay una clara división en tres categorías de tiempos. Por un lado se visualiza a la versión lineal conformando el primer grupo, con el peor tiempo entre los tres grupos, y a su vez extremadamente distante de los otros dos. Por otro lado se tiene en segundo lugar al grupo conformado por las versiones hechas en lenguaje C, las cuales presentan una mejora significativa en tiempo con respecto a la lineal. Dentro de este grupo se aprecia como a mejor tecnología el tiempo disminuye, mas allá de que esta disminución sea de menor magnitud. Finalmente se tiene al tercer grupo conformado por las versiones en lenguaje ASM, el cual no presenta un distanciamiento tan notable con el grupo anterior como con la versión lineal pero, aun así, es otra mejora que cabe destacar. En este grupo se repite el mismo patrón que en el grupo anterior, donde las tecnologías se desempeñan mejor a mayor tamaño del vector.

Podemos afirmar luego de este análisis que nuestra hipótesis no es del todo correcta, porque si bien se respetó el orden propuesto entre las distintas tecnologías cuando se comparaban versiones que compartían el mismo lenguaje, no fue el caso para cuando diferían en el mismo. Se puede concluir con esto que al comparar dos versiones el factor preponderante es el lenguaje y luego es la tecnología.

Aunque ya vimos que nuestra hipótesis es parcialmente correcta, queríamos ver otra métrica para poder medir la eficacia de una implementación. En el caso de los algoritmos que utilizan una matriz de programación dinámica, existe el concepto de **GCUPS** (*Giga Cell Updates Per Second*), que busca medir la eficiencia de una implementación a través de la determinación de cuantas celdas de la matriz se calculan por segundo. Por ejemplo, si tenemos una implementación con 1 GCUPS, eso significa que puede calcular 10^9 celdas en un segundo. En la figura 21 se muestran los resultados del mismo experimento presentado en la figura 20, solo que ahora en lugar del tiempo se muestran los **GCUPS**:



(a) Cantidad de celdas actualizadas por segundo para las distintas implementaciones del algoritmo Needleman-Wunsch en función de las longitudes de las secuencias.



(b) Cantidad de celdas actualizadas por segundo para las distintas implementaciones del algoritmo Needleman-Wunsch en función de la cantidad de celdas en la matriz de programación dinámica.

Figura 21

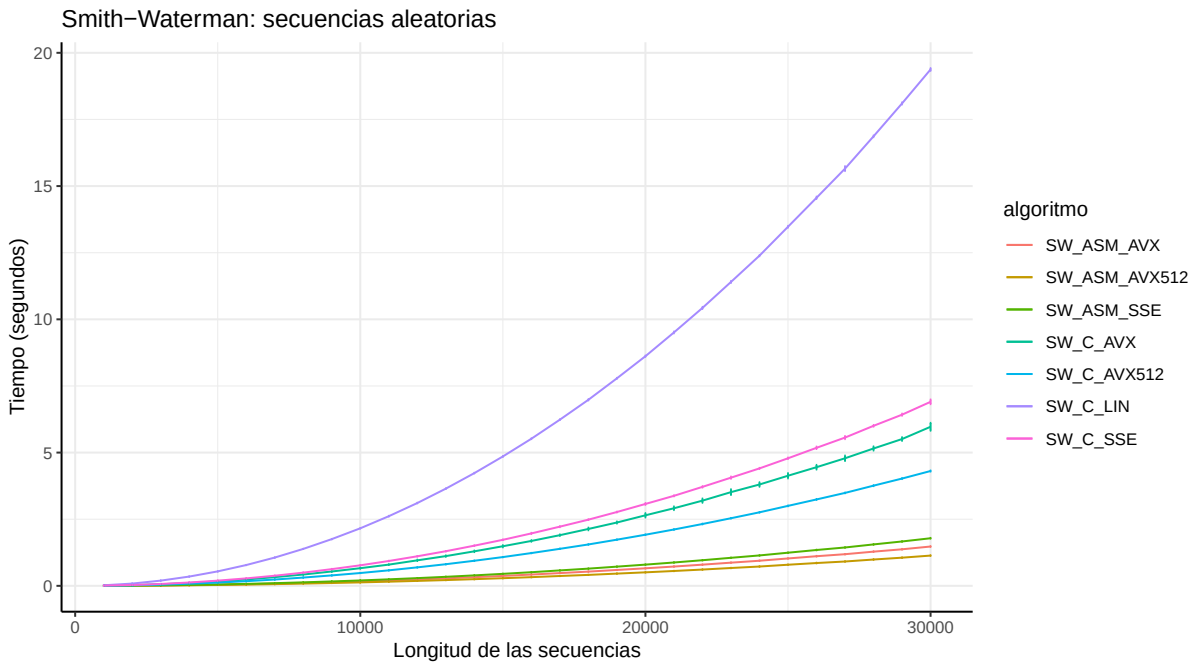
Al igual que en la figura 20, se observa una división en tres grupos. Vemos que las versiones que conforman a cada grupo no cambian y que también se respeta, dentro de cada grupo, el orden según el tamaño del vector de la tecnología involucrada. Una ventaja que presenta esta métrica es que se puede establecer una relación en la velocidad de 2 implementaciones a través de una constante. Por ejemplo, si comparamos la versión NW_C_SSE con la NW_ASM_SSE se puede decir que la segunda está en el orden de 2 veces la velocidad de la primera. Al hablar de estas relaciones también notamos que si vemos

la versión NW_ASM_SSE y la comparamos con NW_ASM_AVX no hay una relación del doble de velocidad, a pesar de que la segunda procesa el doble de celdas en simultaneo de lo que hace la primera debido al tamaño del vector que utiliza. Esto también resulta interesante, porque nos dice que utilizar una mejor tecnología que procesa el doble de datos no viene acompañado de una constante que duplica la cantidad de celdas procesadas por segundo.

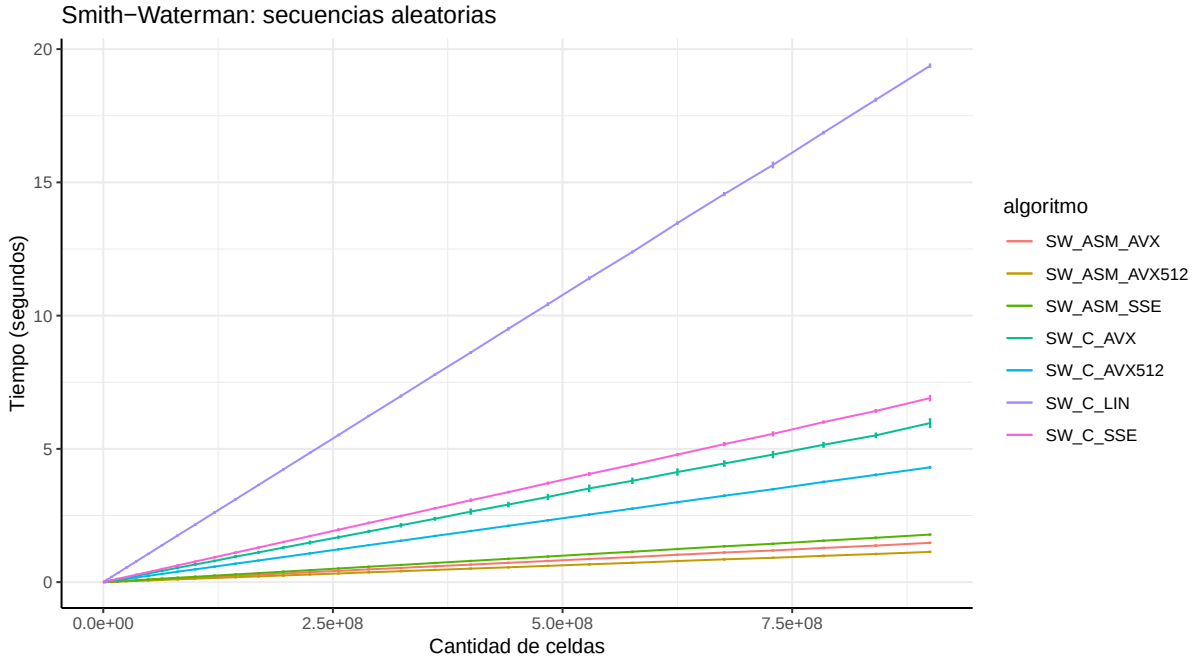
Si nos concentramos en las diferencias de GCUPS que hay entre 2 tecnologías sucesivas dentro de un mismo grupo, vemos que para la del grupo 2, las de lenguaje C, la distancia parece ser la misma en ambos casos, pero en el de lenguaje ASM esto no es así. La versión que hace uso de la tecnologia AVX-512 presenta una diferencia con respecto a AVX que resulta mas amplia que la de AVX contra SSE. Esto puede deberse a que la version de AVX-512 presenta mejoras en las etapas que involucran leer las secuencias, evitando hacer muchas de las operaciones que hacen las otras versiones sin esta tecnologia, y asi logrando un mayor incremento de performance.

Por último vemos que, para todas las versiones, los GCUPS no se ven afectados por un incremento en la cantidad de celdas o las longitudes de las secuencias, sino que siempre se mantiene estable en el mismo valor promedio.

Todo lo analizado hasta ahora fue bajo las implementaciones correspondientes al algoritmo de Needleman-Wunsch, por lo cual falta ver que sucede para las de Smith-Waterman. De manera similar al caso anterior se presentan 2 figuras, la figura 22 muestra el tiempo de ejecución y la figura 23 los GCUPS para las distintas implementaciones del algoritmo Smith-Waterman:

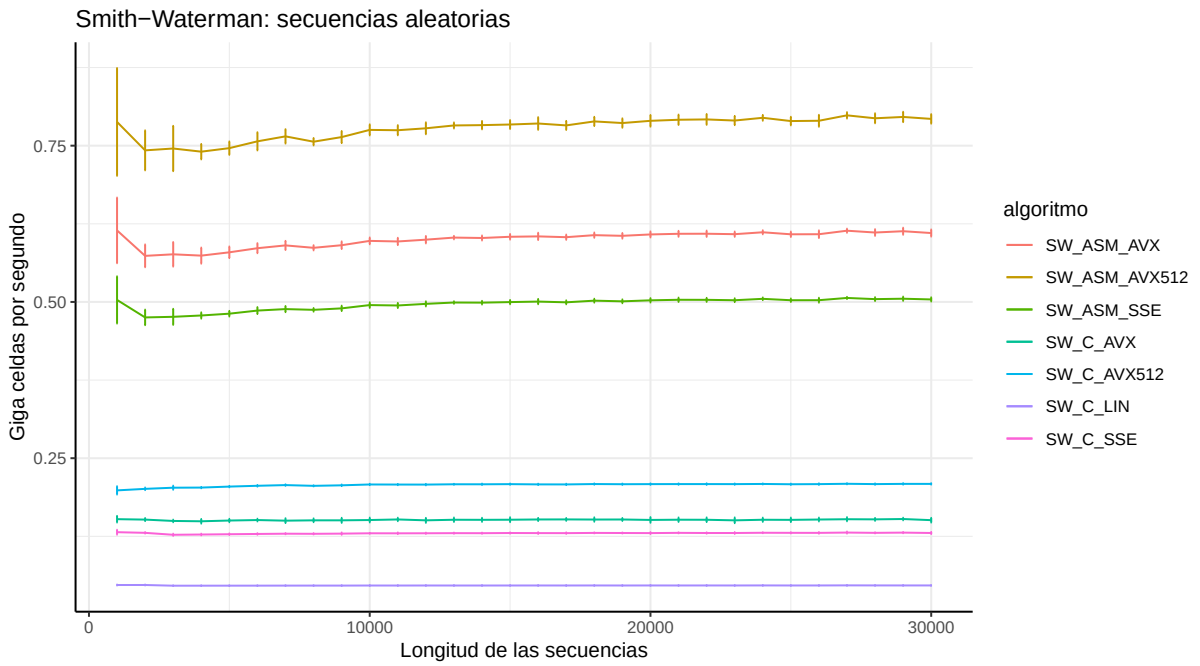


(a) Tiempos de ejecución para las distintas implementaciones del algoritmo Smith-Waterman en función de las longitudes de las secuencias.

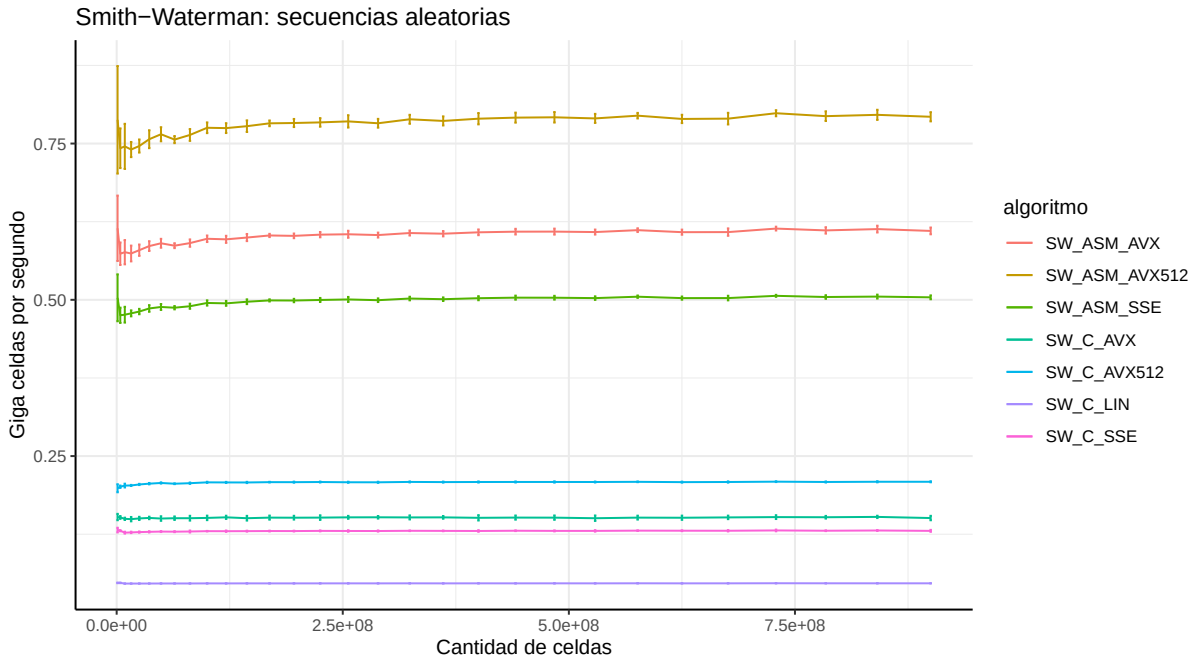


(b) Tiempos de ejecución para las distintas implementaciones del algoritmo Smith-Waterman en función de la cantidad de celdas en la matriz de programación dinámica.

Figura 22



(a) Cantidad de celdas actualizadas por segundo para las distintas implementaciones del algoritmo Smith-Waterman en función de las longitudes de las secuencias.



(b) Cantidad de celdas actualizadas por segundo para las distintas implementaciones del algoritmo Smith-Waterman en función de la cantidad de celdas en la matriz de programación dinámica.

Figura 23: DNA Tomado de Pray 2008

Cabe destacar primero que todo que lo que se observó previamente en el comportamiento del algoritmo Needleman-Wunsch se ve reflejado una vez mas en este caso, mas allá de que el algoritmo presente algunas diferencias en los pasos. Si comparamos los tiempos y los GCUPS de una implementación para los dos algoritmos, notamos que para Smith-Waterman en ambos casos la performance se ve degradada.

Además, otro cambio en este caso es que las diferencias de GCUPS entre las versiones dentro del grupo de lenguaje C en las figuras 23 ya no son equidistantes como lo eran previamente. Esto una vez mas puede deberse con bastante certeza al nuevo paso del algoritmo, el cual degrada la performance para el caso de pasar de usar SSE a AVX, pero no asi cuando se pasa de AVX a AVX-512. Esto último podría deberse a que en AVX-512 las instrucciones utilizadas en este paso extra del algoritmo son más eficientes que en AVX. Finalmente también se puede destacar que la versión en ASM de AVX-512 se degradó menos, y por lo tanto obtuvo una brecha contra la versión ASM de AVX aún más grande en GCUPS que en el caso de Needleman-Wunsch, mientras que la brecha entre las versiones ASM con AVX y SSE se mantuvo igual.

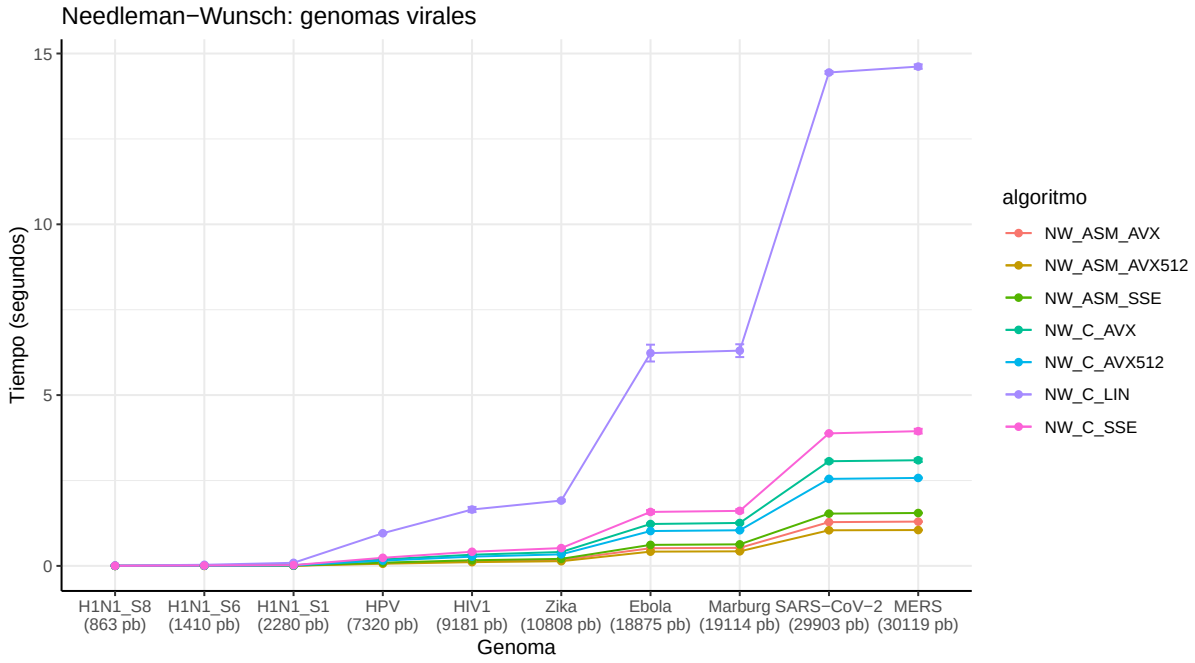
3.2. Secuencias de genomas virales

En este experimento se buscó corroborar si las relaciones entre los desempeños de las distintas implementaciones del algoritmo Needleman-Wunsch obtenidas analizando secuencias aleatorias se mantienen al analizar secuencias reales. Para ello se trabajó con las secuencias de distintos genomas virales o segmentos de los mismos cubriendo un rango de tamaños desde 800 pares de bases hasta 30000 pares de bases aproximadamente.

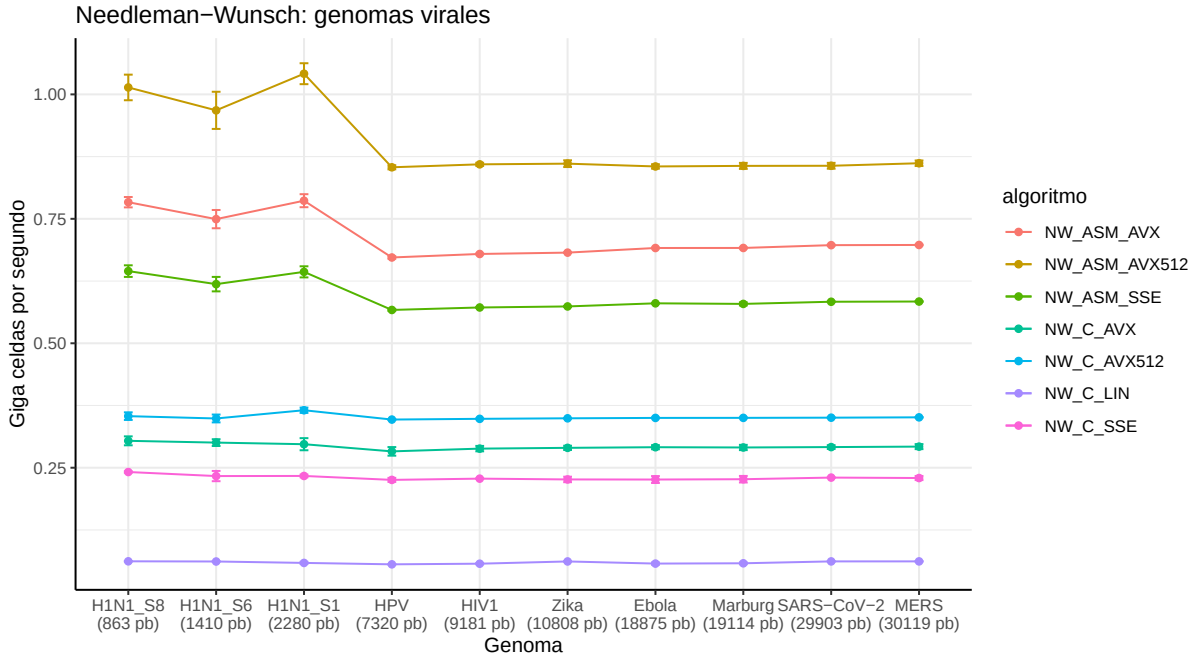
En la figura 24a se muestran los tiempos de ejecución obtenidos al alinear las secuencias de los distintos genomas virales a cada genoma de referencia. Se observa que el tiempo de ejecución crece a medida que lo hace el tamaño del genoma alineado y, al igual que pasa cuando se alinean secuencias aleatorias, los resultados se separan en tres grupos claramente distinguibles. El de peor desempeño solo está constituido por la implementación lineal en lenguaje C mientras que los otros dos grupos contienen las implementaciones que utilizan instrucciones SIMD. Entre estos dos grupos el desempeño es peor en las implementaciones escritas en lenguaje C pero, en ambos casos, el desempeño mejora a medida que se usan tecnologías con mayor tamaño de vector ya que permiten procesar más datos en paralelo.

La cantidad de celdas procesadas por segundo también mostró resultados similares a los obtenidos al alinear secuencias aleatorias (figura 24b). Se distinguen los mismos tres grupos según el desempeño y se observa también que la velocidad de procesamiento se mantiene constante una vez que se alcanza cierto tamaño de secuencias independientemente del tipo de implementación. De igual forma a lo mostrado anteriormente para las secuencias aleatorias, duplicar la cantidad de datos procesados en cada operación SIMD no duplica la velocidad de procesamiento.

Si bien la hipótesis inicial de relaciones entre los desempeños de las distintas implementaciones se cumple de forma parcial, se pueden observar patrones muy similares a los vistos al analizar secuencias aleatorias. Estas relaciones muestran que el factor con más peso sobre el desempeño de las distintas implementaciones es su lenguaje y, en segundo lugar, la tecnología utilizada.



(a) Tiempos de ejecución para las distintas implementaciones del algoritmo Needleman-Wunsch en función de las longitudes de los genomas virales alineados.



(b) Cantidad de celdas actualizadas por segundo para las distintas implementaciones del algoritmo Needleman-Wunsch en función de las longitudes de los genomas virales alineados.

Figura 24

3.3. Simulación de lecturas NGS

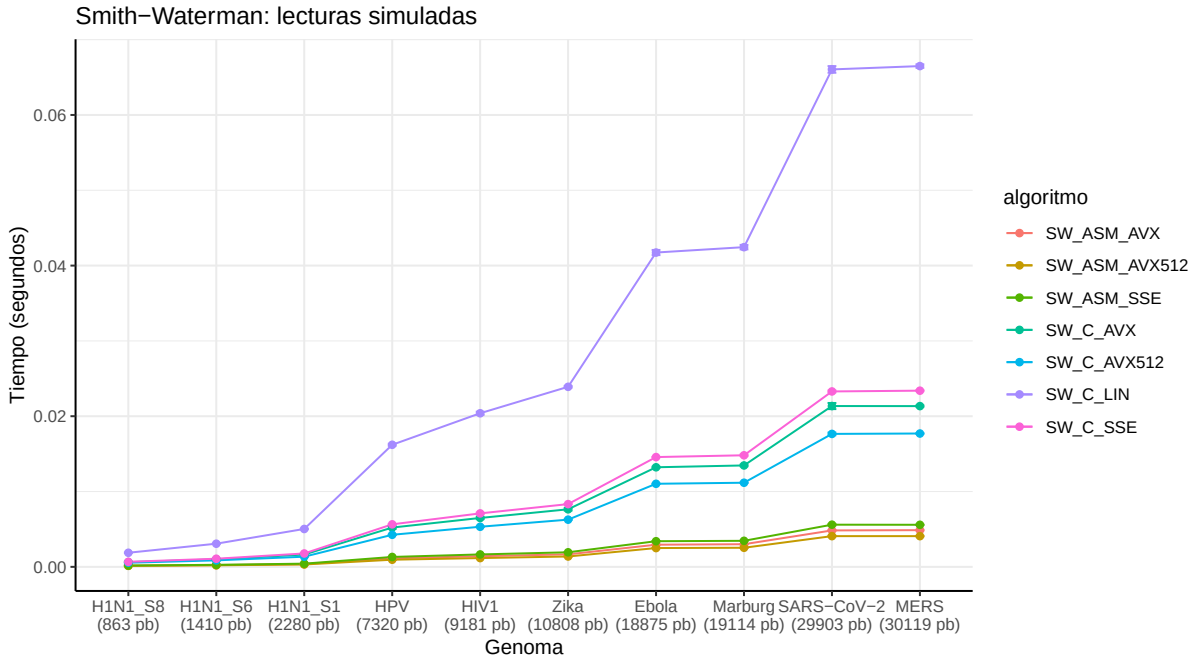
El propósito de este experimento fue comparar los resultados obtenidos para el alineamiento de secuencias aleatorias mediante el algoritmo Smith-Waterman con los resultados obtenidos utilizando el mismo algoritmo para alinear secuencias que simulan ser lecturas provenientes de un secuenciador de

próxima generación (NGS) Illumina. Para ello se generaron lecturas artificiales a partir de cada uno de los genomas virales de referencia con características similares a las reales y un largo de 100 pares de bases, que luego fueron alineadas al genoma de referencia que les dio origen.

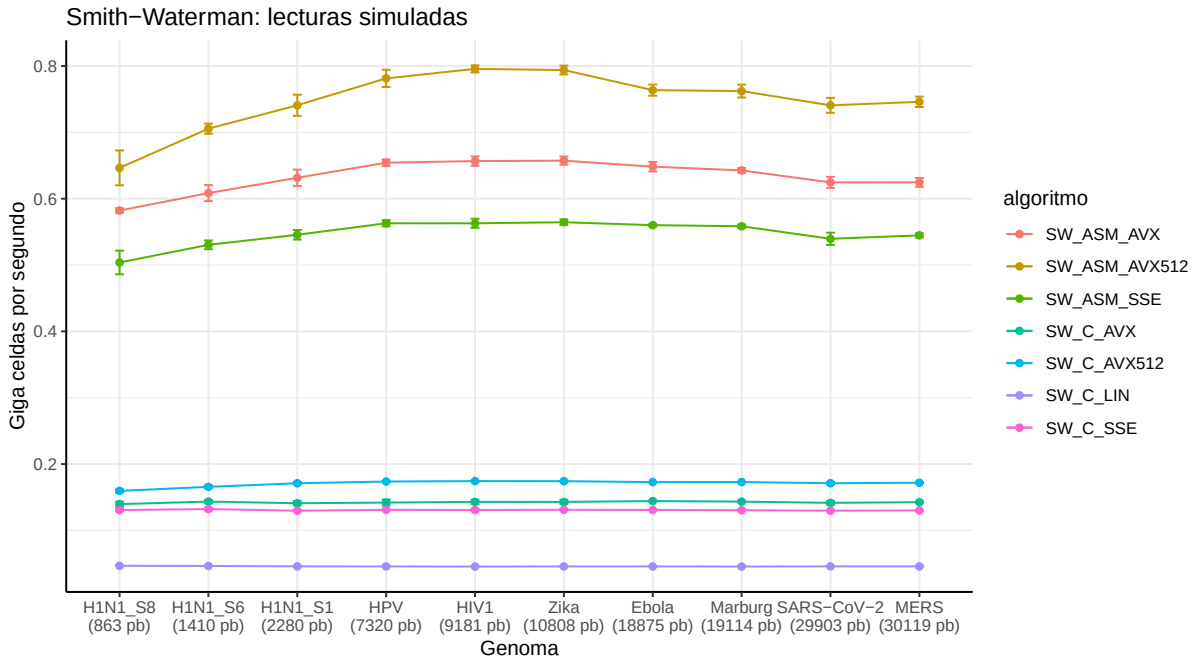
En la figura 25a se muestran los tiempos de ejecución obtenidos al alinear las lecturas simuladas a cada uno de los genomas de referencia. De forma similar a lo sucedido al ejecutar el algoritmo Smith-Waterman con secuencias aleatorias, se observan tres grupos claramente diferenciables respecto de su desempeño. El grupo que más tiempo tarda está integrado por la implementación en C de forma lineal y luego lo suceden los grupos que emplean instrucciones SIMD. Entre estos dos grupos el desempeño es claramente superior para las versiones desarrolladas en ASM. Adicionalmente, dentro de cada grupo, el tiempo de ejecución es menor a medida que aumenta el tamaño de vector utilizado para procesar los datos.

Respecto de la velocidad de procesamiento, también se observa una tendencia similar a lo sucedido con las secuencias aleatorias (figura 25b). La cantidad de celdas actualizadas por unidad de tiempo se mantiene constante al aumentar el tamaño de las secuencias alineadas, salvo cuando se trabaja con secuencias pequeñas. Esto permite observar las mismas relaciones entre los tipos de implementaciones y corroborar que, si bien la utilización de vectores del doble de tamaño mejora el desempeño, no logra duplicar la velocidad de procesamiento.

Las conclusiones de este experimento resultan entonces similares a las del alineamiento de secuencias aleatorias mediante el algoritmo Smith-Waterman. Si bien procesar más datos en paralelo aumentando el tamaño del vector SIMD permite mejorar el desempeño, el factor más determinante es el lenguaje de la implementación.



(a) Tiempos de ejecución para las distintas implementaciones del algoritmo Smith-Waterman en función de las longitudes de los genomas virales a los cuales se alinearon las lecturas simuladas.

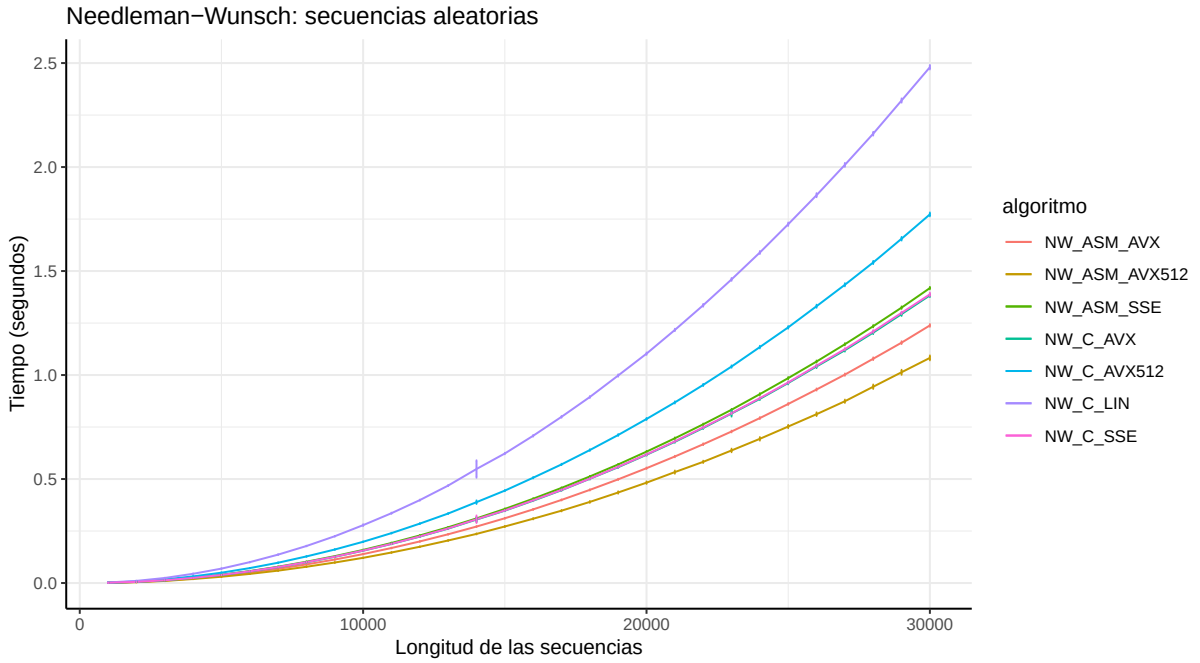


(b) Cantidad de celdas actualizadas por segundo para las distintas implementaciones del algoritmo Smith-Waterman en función de las longitudes de los genomas virales a los cuales se alinearon las lecturas simuladas.

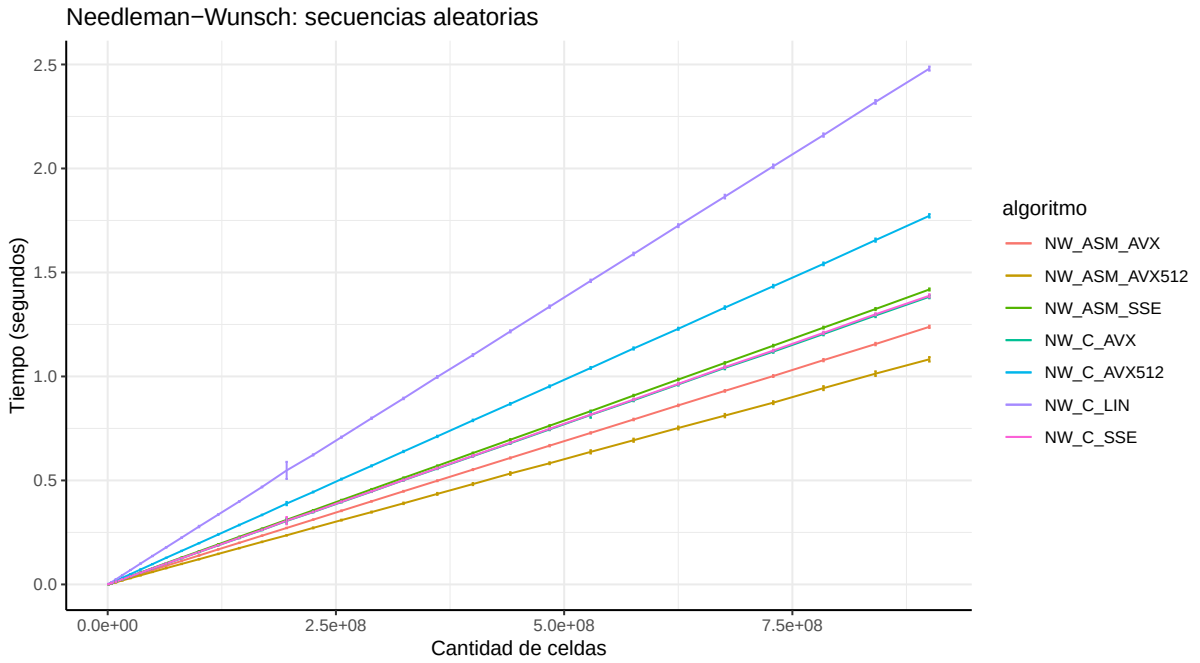
Figura 25: DNA Tomado de Pray 2008

3.4. Secuencias aleatorias O3

Presentamos primero, en la figura 26, los resultados para las versiones del algoritmo de Needleman-Wunsch donde se gráfica el tiempo de ejecución en función de la longitud de las secuencias y de la cantidad de celdas en la matriz de programación dinámica respectivamente:



(a) Tiempos de ejecución para las distintas implementaciones del algoritmo Needleman-Wunsch en función de las longitudes de las secuencias.



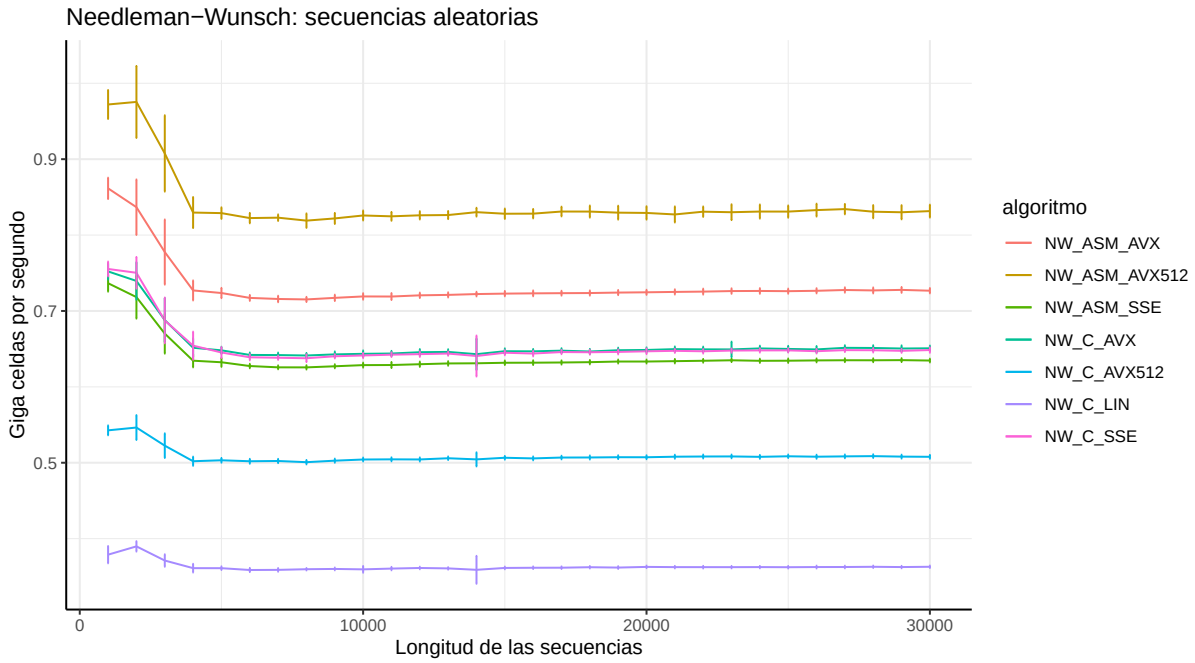
(b) Tiempos de ejecución para las distintas implementaciones del algoritmo Needleman-Wunsch en función de la cantidad de celdas en la matriz de programación dinámica.

Figura 26

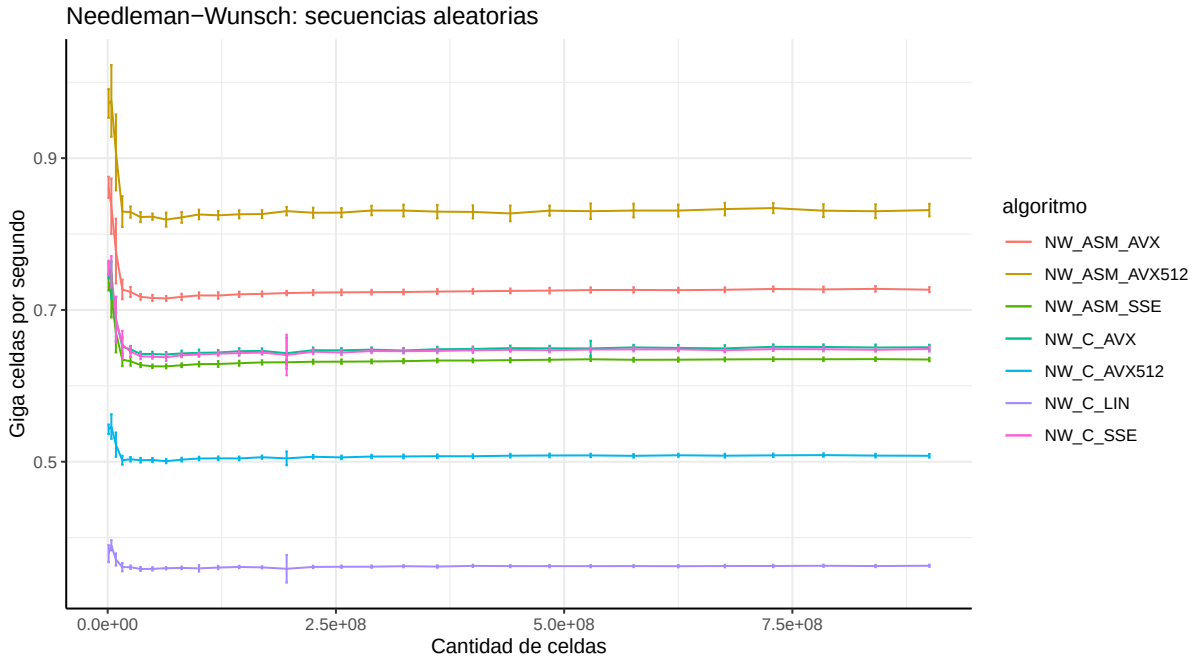
Un primer aspecto que notamos al observar tanto la figura 26a como la 26b es que se pierde la clara división en grupos de las implementaciones como pasaba en el experimento 1. Este fenómeno se debe a las optimizaciones $O3$ y termina produciendo una reducción, en la mayoría de los casos, de la brecha de performance que existe entre distintas versiones sin aplicar optimizaciones al compilar. Se puede ver que en general se pudo optimizar el rendimiento gracias a la compilación con optimización. Podemos remarcar el caso de la versión lineal en *C*, donde en el Gráfico 20b presentaba una cota máxima de 15 segundos aproximadamente y utilizando la optimización tenemos una de 2.5 segundos.

Sin embargo, no siempre se obtuvo el mejor de los resultados a través de estas optimizaciones. Este es el caso de la versión AVX512 hecha en *C*, donde las optimizaciones la dejan atrás con respecto a las demás. Esto podría atribuirse a que el compilador no esta preparado para optimizar AVX512, como sucede con las versiones en SSE y AVX.

A continuación mostramos una vez más como son los *GCUPS* en función de la longitud de las secuencias y de la cantidad de celdas en la matriz de programación dinámica:



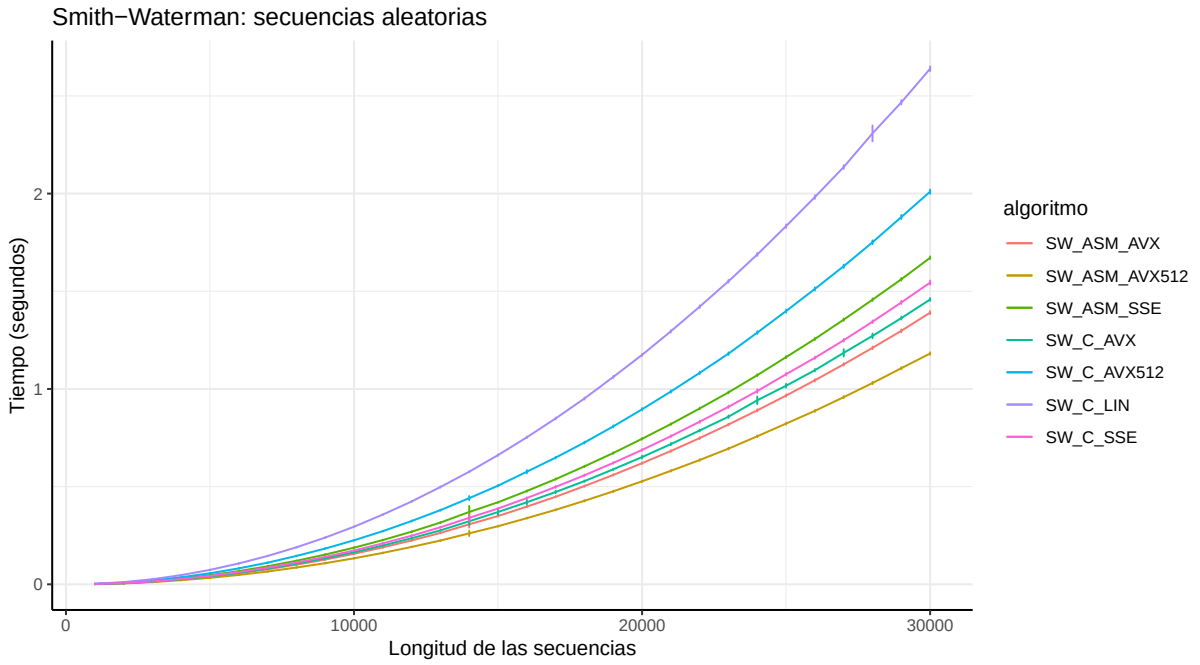
(a) Cantidad de celdas actualizadas por segundo para las distintas implementaciones del algoritmo Needleman-Wunsch en función de las longitudes de las secuencias.



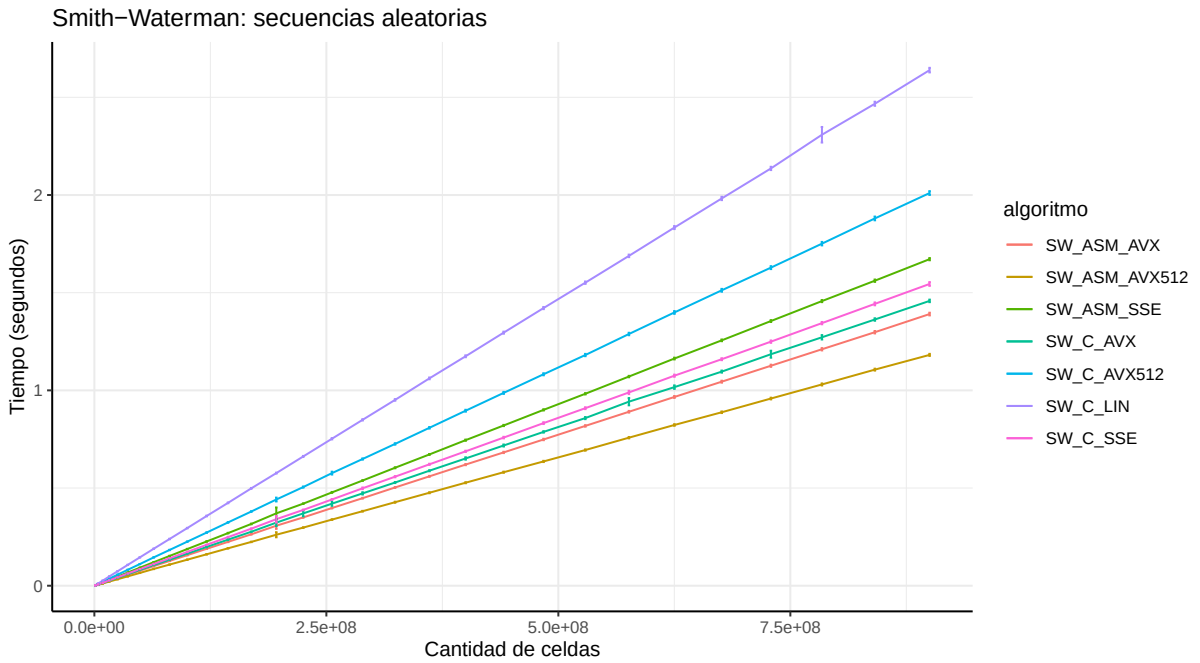
(b) Cantidad de celdas actualizadas por segundo para las distintas implementaciones del algoritmo Needleman-Wunsch en función de la cantidad de celdas en la matriz de programación dinámica.

Figura 27

Observando el Gráfico 27b podemos ver que la optimización funciona particularmente bien para el caso de SSE donde su versión en *C* supera a la de ASM, como se esperaba. Por otro lado, si miramos la versión en AVX ocurre un comportamiento similar al señalado con AVX512 pero en menor medida, y mas aún, es idéntico al de la versión SSE en *C*. Además podemos otra diferencia de los gráficos para secuencias aleatorias sin optimización con los que sí la implementan. En los valores mas pequeños de longitud de secuencias o cantidad de celdas, se obtiene su "pico" de velocidad para luego ir descendiendo.



(a) Tiempos de ejecución para las distintas implementaciones del algoritmo Smith-Waterman en función de las longitudes de las secuencias.



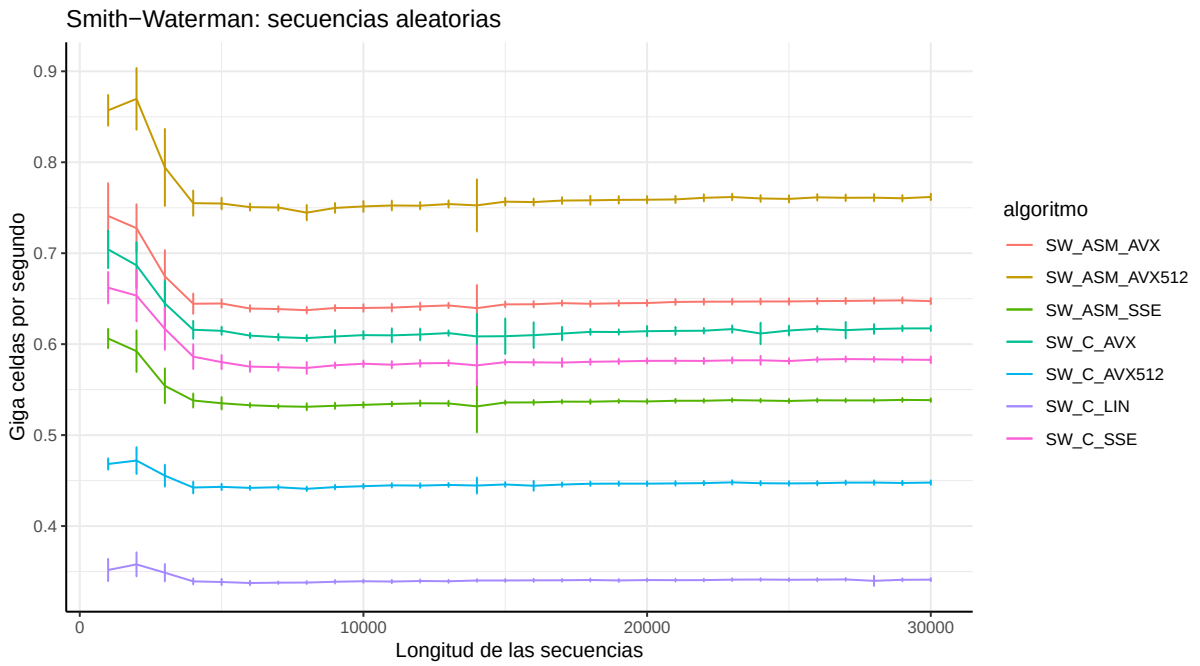
(b) Tiempos de ejecución para las distintas implementaciones del algoritmo Smith-Waterman en función de la cantidad de celdas en la matriz de programación dinámica.

Figura 28

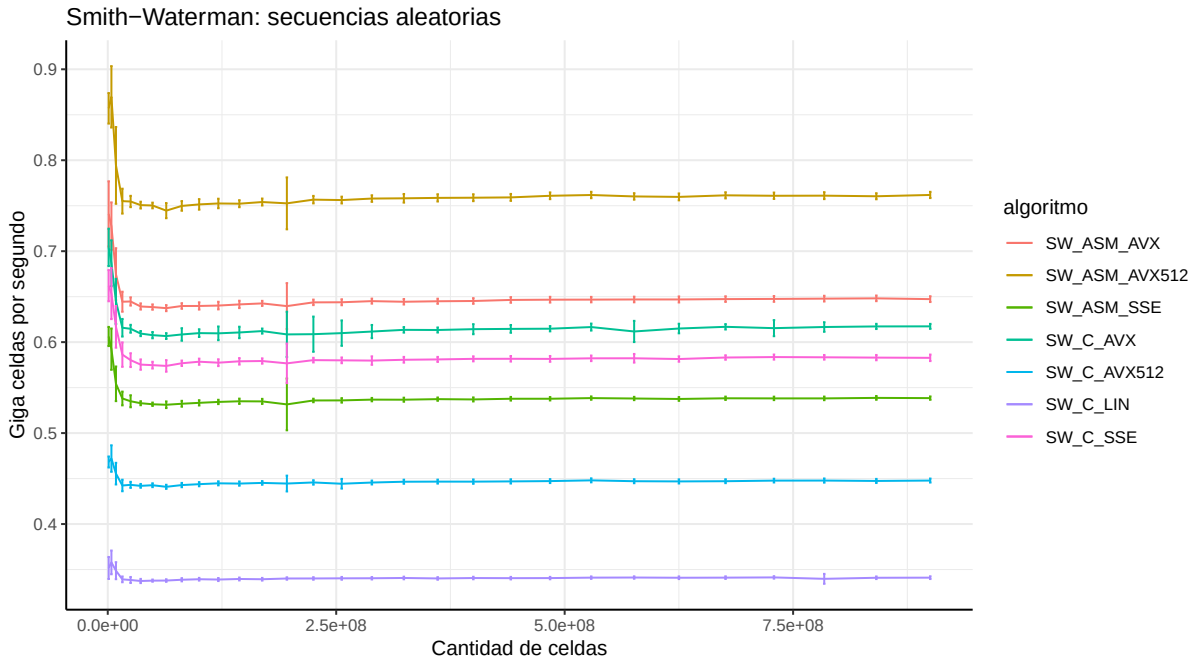
Ahora resta ver que sucede para las mismas instancias en las implementaciones optimizadas con $O3$ del algoritmo Smith-Waterman. De manera similar al caso anterior, se presentan 4 figuras. Las figuras 28b y 28a muestran el tiempo de ejecución en función de la longitud de las secuencias y de la cantidad de celdas en la matriz de programación dinámica y, las figuras 29b y 29a, los GCUPS en función de la longitud de las secuencias y de la cantidad de celdas en la matriz de programación dinámica.

En la figura 28 se visualiza una degradación en el tiempo con respecto a la versión de Needleman una vez mas, debido al branching adicional introducido por el código nuevo del Smith-Waterman. Podemos ver también que ahora se estableció una clara diferencia entre las tres versiones que antes se encontraban equiparadas. Si bien la versión de SSE en ASM sigue presentando el peor resultado entre estas, ya no se encuentran a la par las versiones en C con AVX y SSE, sino que la de AVX es mejor y se asemeja a su versión en ASM. Adicionalmente, también se observa en el caso de este algoritmo que la implementación en C que contiene instrucciones AVX-512 no es optimizada de forma satisfactoria por el compilador.

En los gráficos de la figura 29 vemos que las curvas de velocidad de procesamiento no se encuentran solapadas como en 27b. Esto se puede atribuir a la única diferencia significativa que posee este algoritmo, la cual es la forma de tomar el máximo de los valores actuales junto con su posición. Esto demuestra como una función puede afectar tan significativamente a la performance del algoritmo según como esté implementada.



(a) Cantidad de celdas actualizadas por segundo para las distintas implementaciones del algoritmo Smith-Waterman en función de las longitudes de las secuencias.



(b) Cantidad de celdas actualizadas por segundo para las distintas implementaciones del algoritmo Smith-Waterman en función de la cantidad de celdas en la matriz de programación dinámica.

Figura 29

4. Discusión

En todos los experimentos, trabajar con vectores SIMD del doble de tamaño generó mejoras en el desempeño pero no logró duplicar la velocidad de procesamiento, o lo que es equivalente, reducir el tiempo de procesamiento a la mitad. Existen varios motivos que podrían causar el comportamiento observado. Por un lado, si bien las instrucciones sobre vectores de mayor tamaño incrementan la cantidad de celdas que se procesan en paralelo, tienen mayor latencia asociada. Adicionalmente, para determinadas instrucciones no existe una operación equivalente que procese el doble de datos, entonces es necesario utilizar otras estrategias que pueden requerir otras instrucciones y, en algunos casos, la utilización de varias instrucciones para obtener el mismo comportamiento. Por otro lado, todas las implementaciones tienen asociado un tiempo constante correspondiente al backtracking que no cambia. Finalmente, cada operación SIMD está compuesta por varias micro-operaciones que son ejecutadas fuera de orden y podría resultar más complejo optimizar la ejecución de instrucciones que emplean vectores de mayor tamaño porque poseen un mayor número de dependencias. Todos estos factores podrían explicar porque no se duplica la performance cuando se duplica el tamaño del vector empleado.

Por otro lado, al aplicar optimizaciones del compilador para las implementaciones en lenguaje C, se observó que el desempeño para secuencias de tamaño pequeño es mejor que el promedio. Esto no pasaba o al menos era menos marcado al no utilizar optimizaciones. El comportamiento observado podría deberse a que el código optimizado funciona mejor para instancias chicas del problema, pero es más probable que se deba a la diferencia entre las instancias que estamos utilizando. En las especificaciones que se detallan en la sección 3 se puede ver que la instancia utilizada en este experimento tiene un tamaño de cache nivel 3 mayor y un clock con mayor frecuencia. Esto produce tanto una mejora en el tiempo como en las GCUPS en general. Pero, el mayor impacto sobre las instancias chicas del problema puede ser explicado por la posibilidad de mantener una mayor parte o incluso la totalidad de la matriz de programación dinámica en la cache .

Finalmente, el desempeño del algoritmo Smith-Waterman fue peor que el Needleman-Wunsch en todas las condiciones experimentales. Esto probablemente se debe al hecho de que se agrega un paso extra en cada iteración para obtener el valor máximo. Este paso inevitablemente agrega mas instrucciones y, además, introduce otro caso de branching para decidir si actualizar o no la posición máxima. En esta situación, el camino a tomar no es fácil de predecir porque no depende de la iteración actual, sino de los puntajes previos. En un peor escenario, podría ocurrir que se encuentre un nuevo máximo cada dos iteraciones, lo que provocaría que se produzcan saltos cada dos iteraciones. Por lo tanto, siempre fallaría la predicción, haciendo inútil al sistema de *branch prediction*. Sin embargo, la versión de AVX512 en ASM fue la que menos degradación obtuvo. Suponemos que esto se puede atribuir al mayor procesamiento de datos con respecto a las otras versiones, el cual reduce las iteraciones del algoritmo significativamente y, a su vez, hace que se eviten evaluaciones que pueden producir branching adicional.

5. Conclusiones y trabajo futuro

Al aplicar los conocimientos obtenidos en la materia, particularmente sobre el modelo de ejecución SIMD, corroboramos que vale la pena paralelizar este tipo de algoritmos de alineamiento de secuencias biológicas que hacen uso de técnicas de programación dinámica. Por otro lado vimos que paralelizar no solo ayuda a trabajar con mas datos en simultáneo, sino que también aporta otro tipo de ventajas en la

ejecución, como la reducción de accesos a memoria y el branching.

Los dos algoritmos de alineamiento de secuencias utilizados en este trabajo son similares en varías partes, pero el de Smith-Waterman posee un paso extra. Este paso adicional puede parecer despreciable con respecto a toda la implementación pero, en los resultados, se puede observar el impacto que genera en la performance global del algoritmo. Esto demuestra como un buen diseño sobre un único paso es crítico para todo el desempeño.

Otro aspecto a considerar es la aplicación de optimizaciones a través del compilador para implementaciones que difieren en el tamaño de vector SIMD empleado, donde en el caso general uno espera que estas presenten una mejora en la performance pero mantengan la diferencia relativa entre sus desempeños. Pudimos verificar que esto no siempre se cumple, como en el caso de comparar el rendimiento de dos implementaciones hechas en C, una con SSE y otra con AVX-512. En este caso lo que ocurrió fue que al aplicar las optimizaciones la primera sobrepasó ampliamente a la segunda, sugiriendo que todavía no hay buenos métodos de optimización implementados en el compilador para el set de instrucciones AVX-512.

A futuro sería interesante explorar otros algoritmos que forman parte del proceso de secuenciación de genomas para poder introducir donde sea posible la paralelización a nivel de datos. Por otro lado, también podría llevarse a cabo un análisis para encontrar alternativas a las implementaciones realizadas en este trabajo. Por ejemplo, nuevas implementaciones podrían obtenerse a partir de las actuales, modificando las instrucciones utilizadas en las distintas etapas de cada una de estas. Otra alternativa interesante sería proponer una estrategia distinta a la planteada por nosotros para llevar a cabo la paralelización del acceso a la matriz de programación dinámica, en busca de mejoras en el uso de la memoria, la cache, etc, para así poder incrementar el rendimiento final del algoritmo.

Finalmente, otro factor a considerar es la limitación que conlleva trabajar con cada tipo de dato, lo cual limita el tamaño de las secuencias con las que podemos tratar. En nuestras implementaciones estamos limitados a secuencias de a lo sumo 32768 pb por trabajar con un tipo de dato de 2 bytes (word). Considerando esto, proponemos realizar una implementación para distintos tamaños de tipo de dato, logrando así, un algoritmo más flexible que elija el menor tamaño de tipo de dato necesario dado el tamaño de las secuencias a alinear.

6. Bibliografía

Alberts, Bruce y col. (sep. de 2017). *Molecular Biology of the Cell*. Ed. por John Wilson y Tim Hunt. Vol. 40. 9. Garland Science, pág. 1709. ISBN: 9781315735368. DOI: 10.1201/9781315735368. URL: https://www.taylorfrancis.com/books/9781317563754.

Johnson, Irving S (1983). «Human Insulin from Recombinant DNA Technology Author (s): Irving S . Johnson». En: *Science* 219.4585, págs. 632-637.

Miller, Webb (2006). *An Introduction to Bioinformatics Algorithms*. Vol. 101. 474, págs. 855-855. ISBN: 0262101068. DOI: 10.1198/jasa.2006.s110.

Needleman, Saul B. y Christian D. Wunsch (1970). «A general method applicable to the search for similarities in the amino acid sequence of two proteins». En: *Journal of Molecular Biology* 48.3, págs. 443-453. ISSN: 00222836. DOI: 10.1016/0022-2836(70)90057-4.

Pray, Leslie A. (2008). «Discovery of DNA Structure and Function: Watson and Crick». En: *Nature Education*. URL: https://www.nature.com/scitable/topicpage/discovery-of-dna-structure-and-function-watson-397.

Smith, T.F. y M.S. Waterman (mar. de 1981). «Identification of common molecular subsequences». En: *Journal of Molecular Biology* 147.1, págs. 195-197. ISSN: 00222836. DOI: 10.1016/0022-2836(81)90087-5. URL: https://linkinghub.elsevier.com/retrieve/pii/0022283681900875.