Filtrado eficiente de secuencias de ADN

Zaina Salym

Resumen– La secuenciación genómica es un avance científico clave que permite decodificar secuencias de ADN, detectar variaciones genéticas y diagnosticar enfermedades. En la actualidad existen una gran variedad de algoritmos que permiten encontrar mutaciones en secuencias genómicas. La mayoría de estos algoritmos de comparación de secuencias se basan en métodos de programación dinámica. Estos algoritmos presentan costes computacionales prohibitivos para la cantidad masiva de datos producida por los secuenciadores. El objetivo de este proyecto es acelerar los algoritmos de alineamiento de secuencias utilizando un método de pre-filtrado basado en el conteo de k-mers.

Palabras clave– Alineamiento, filtro, ADN, kmers, bioinformática.

Abstract– Genomic sequencing is a key scientific advance that allows decoding DNA sequences, detecting genetic variations, and diagnosing diseases. Nowadays, there is a great variety of algorithms that allow finding mutations in genomic sequences. Most of these algorithms for sequence comparison are based on dynamic programming methods. These methods present prohibitive computational costs in order to cope with the massive amount of data produced by modern sequencers. The goal of this project is to accelerate sequence alignment algorithms by using a pre-filtering method based k-mers counting.

Keywords– Alignment, filters, DNA, kmers, bioinformatics

1 Introducción

La secuenciación del genoma humano [Cla+10] representa uno de los avances más importantes de las últimas décadas. El objetivo es obtener un genoma de referencia que represente las características comunes de toda la especie humana. Este genoma de referencia permite estudiar los efectos que provocan las variaciones que se producen en los genes de algunos individuos y pueden causar enfermedades. Este suceso se denomina mutación [Loe] y se produce de manera esporádica. Estas mutaciones son las que nos hacen diferentes ya que determinan todas nuestras características (p. ej. el color de la piel, el color de ojos o la estatura). Sin embargo, a veces una mutación puede alterar las «instrucciones» para fabricar las proteínas. Esta alteración puede llegar a provocar un fallo en el funcionamiento de las células y puede causar una enfermedad genética.

La bioinformática[LGG01] es la disciplina científica que se dedica a analizar el genoma con el fin de encontrar los mecanismos de evolución, la causas de los problemas biológicos y otras cuestiones relacionadas. Esta disciplina procesa un gran cantidad de datos ya que en este contexto se manejan colecciones muy grandes de ADN y ARN. Esto supone un desafío computacional muy importante. No obstante, pese a ser colecciones de datos muy grandes, las secuencias son muy altamente similares entre sí. Es decir, dos genomas humanos pueden tener una similitud mayor al 99,9 %. Por esta razón, normalmente solo es necesario procesar ciertas regiones específicas del genoma. Para determinar si una secuencia pertenece a una región u otra se suelen utilizar algoritmos de alineamiento que computan el grado de similitud entre las secuencias. Los más utilizados son Smith-Waterman [SW+81] (SW) y Needleman-Wunsch [NW70] (NW).

El algoritmo NW realiza un alineamiento global de dos secuencias y garantiza encontrar la puntación máxima. Esta puntuación se especifica en una matriz de similitud que contiene una columna para cada carácter de la secuencia A y una fila para cada carácter de la secuencia B. De este modo, si estamos alineando secuencias de tamaño n y m, la complejidad del algoritmo es $O(nm)$. El algoritmo SW es una variación de este último, ya que en vez de realizar un alineamiento global, realiza alineamientos locales para determinar las regiones similares entre dos secuencias.

Pese a que estos dos algoritmos devuelven el grado de similitud exacto entre dos secuencias, se basan en métodos

• E-mail: zaina.salym@e-campus.uab.cat
• Mención: Ingeniería de Computadores
• Tutor: Santiago Marco-Sola, Juan Carlos Moure (Departamento de Arquitectura de Computadores y Sistemas Operativos)
• Curso 2019/20

de programación dinámica que incurren en costes computacionales muy altos (complejidad cuadrática). No obstante, cuando dos secuencias son muy diferentes entre sí, no merece la pena invertir recursos para alinearlas y posteriormente descartarlas por ser muy divergentes. Es preferible utilizar algoritmos de pre-filtrado para detectar si son lo suficientemente parecidas como para ser alineadas. Por este motivo, el objetivo principal de este trabajo es analizar y optimizar un algoritmo de filtrado genético basado en el conteo de k-mers[MS19]. De este modo, se pretende acelerar el rendimiento de los algoritmos de alineamiento y así ayudar a detectar variaciones y mutaciones genéticas de un manera más eficiente.

Cabe destacar que la mayoría de los algoritmos usados en el ámbito de la bioinformática son actualmente viables gracias a la evolución que han tenido las arquitecturas de computadores. La introducción de técnicas de paralelización en CPU, aceleración en GPU, uso de memória Cache y el desarollo de herramientas que permiten explotar todo Hardware son factores clave que permiten la viabilidad de estos algoritmos. Para abordar las limitaciones que presenta el algoritmo de pre-filtrado, se aplicarán estas técnicas para mejorar el rendimiento de la aplicación.

2 DESCRIPCIÓN DEL ALGORITMO

El término de k-mer[Bur+99; Par15] se refiere al conjunto de subsecuencias de longitud k que pueden aparecer dentro de una secuencia. Una secuencia de longitud L contiene $L - k + 1$ k-mers de un total de n^k k-mers posibles, donde n es el número de caracteres del alfabeto usado (p. ej., en el caso del ADN, $n = 4$).

El algoritmo de pre-filtrado cuenta la cantidad de veces que aparece cada k-mer en la secuencia y guarda el resultado en un histograma. A diferencia de los algoritmos de alineamiento explicados anteriormente, en lugar de comparar dos secuencias sobre una estructura bidimensional, se comparan estos histogramas sobre una estructura lineal, factor clave que favorece el rendimiento de manera significativa. La longitud de la k es un parámetro importante para determinar, ya que si esta es muy pequeña habrá mucha redundancia de k-mers en las subcadenas y por lo tanto el filtro perderá precisión. Sin embargo, si la longitud de k es muy grande, el histograma correspondiente será muy grande y puede llegar a penalizar el rendimiento [CM14]. Para entender mejor el funcionamiento del algoritmo supongamos el siguiente ejemplo:

- Supongamos que tenemos la siguiente cadena T perteneciente al genoma de referencia.

A	G	C	A	T	C	A

- Supongamos que tenemos la siguiente muestra P perteneciente a un paciente.

G	A	G	A	T	C	A

- Por último, supongamos que k=2.

La longitud de P es 7, por lo tanto, tendrá 6 k-mers $(7 - 2 + 1)$ de 16 (4^2) k-mers posibles. Nuestro histograma será una tabla de 16 posiciones. El conteo se realiza de la siguiente manera:

- Se realizan dos histogramas, uno para T y otro para P tal y como se muestra en las figuras 1 y 2:

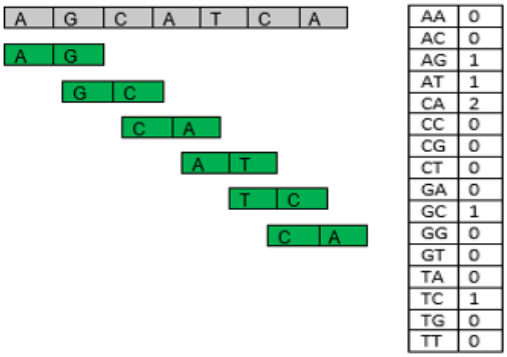

Fig. 1: Histograma de T

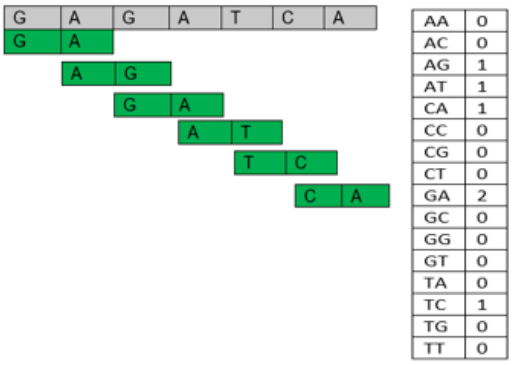

Fig. 2: Histograma de P

- Se comparan los dos histogramas a partir de la distancia que hay entre ellos

Teóricamente la distancia del k-mer se calcula con la siguiente ecuación 1.

$$Dist_{k-mer}(p,t) = \sum_{u \in \Sigma^k} |N(P,u) - N(t,u)| \quad (1)$$

El resultado de esta ecuación es una cota mínima del número de diferencias entre las dos cadenas. En este ejemplo, la distancia calculada entre GAGATCA y AGCATCA es 4. Es importante destacar que el algoritmo usado en este proyecto no filtra en función de la distancia exacta de alineamiento. Los algoritmos de pre-filtrado aproximan una cota mínima de esta distancia y permiten descartar pares cuya distancia de alineamiento excede el máximo grado de divergencia tolerado (p. ej. 5 %). De este modo, podemos descartar rápidamente aquellas secuencias altamente divergentes (TP - «True Positives»), reduciendo el número total de secuencias a analizar. Dado que el algoritmo calcula una cota mínima, en algunas ocasiones el método no podrá descartar correctamente secuencias muy divergentes (FP - «False Positives») de las verdaderamente similares (TN - «True Negatives»). No obstante, el método nunca incurre en error («sin

pérdida» o «lossless»), dado que nunca descarta un par de secuencias altamente similares entre si ($FN = 0$, «False Negatives»). Es decir, el filtro nunca pierde datos «buenos». De esta forma y para evaluar la precisión de nuestro filtro, definimos la suma de TP y TN como «Hits».

3 OBJETIVOS

Tal como se ha explicado anteriormente, el objetivo principal de este trabajo consiste en optimizar el algoritmo de filtrado con el fin de alinear las secuencias de manera más eficiente. A partir de esto se plantean los siguientes objetivos:

1. Investigar y entender el funcionamiento de los algoritmos de comparación de secuencias más usados en el ámbito de la bioinformática y detectar los cuellos de botella que presentan.
   1.1. Smith-Waterman [SW+81].
   1.2. Needleman-Wunsch [NW70].
   1.3. LCS o Levenshtein (Distáncia de edición).
2. Entender el algoritmo de filtrado de secuencias y analizar con la implementación actual.
3. Establecer un conjunto de datos de prueba representativos del problema a resolver.
4. Caracterizar la aplicación en un procesador Intel Skylake[Wik] y ARMv8[Ryz06].
   4.1. Determinar los cuellos de botella (bottlenecks) para detectar las limitaciones de nuestro algoritmo (memoria y cómputo).
   4.2. Realizar un benchmark inicial para determinar la selección óptima del parámetro k óptima en precisión (para el conjunto de datos de prueba seleccionado).
5. Analizar e interpretar los resultados de la caracterización para proponer optimizaciones de rendimiento.
   5.1. Optimizar el algoritmo para mejorar la eficiencia del algoritmo.
   5.2. Diseñar e implementar versiones paralelas para CPU (OpenMP) y GPU (CUDA o OpenACC).
6. Comparar los resultados obtenidos con los producidos por otros algoritmos.

4 METODOLOGÍA

Para llevar a cabo este proyecto se ha aplicado una metodología ágil. Esta metodología proporciona un modelo iterativo e incremental susceptible a cambios y flexible para incorporar los resultados obtenidos en desarrollo. Para lograr cada uno de los objetivos se han realizado reuniones semanales con los tutores para realizar un «feedback» sobre las tareas realizadas en la semana en curso y definir las próximas tareas a realizar. Esto permite tener un control sobre la evolución del proyecto y detectar los problemas que puedan surgir con antelación. El proyecto se ha subdividido en distintas fases:

1. Recopilación de información. Esta es una de las fases más importantes del proyecto ya que se realiza una investigación para entender el contexto del problema. También se realiza un análisis de las soluciones propuestas por otros investigadores.
2. Caracterización de la aplicación. Se realiza un análisis de rendimiento del algoritmo de filtrado base para detectar los posibles cuellos de botella. En esta fase se analizan las posibles soluciones a implementar y las herramientas a usar.
3. Implementación de versiones más eficientes. Se aplicarán distintas técnicas para desarrollar versiones más eficientes del código a partir de las soluciones propuestas en la fase anterior.
4. Analizar los resultados obtenidos. Se realiza un análisis de los resultados obtenidos para cuantificar las mejoras obtenidas. En caso de no obtener los resultados de rendimiento se debe replantear la fase anterior.
5. Comparativa y análisis de resultados. Se compararán todos los resultados obtenidos con con los producidos por otros algoritmos para cuantificar la mejora obtenida.

5 ANÁLISIS INICIAL DEL ALGORITMO

En este apartado se realizará un análisis de rendimiento del algoritmo con distintos parámetros y en distintas arquitecturas. El objetivo es detectar los principales cuellos de botella y analizar el comportamiento del algoritmo en dos arquitecturas distintas (ARM y Intel).

5.1. Parámetros de entrada

Los parámetros de entrada son los siguientes:

- Algorithm: Parámetro que permite elegir el algoritmo a usar para realizar el alineamiento.
  Existen tres opciones:
  - kmer-filter: Algoritmo de filtrado basado en el conteo de k-mers.
  - edit-dp: Algoritmo basado en programación dinámica para calcular la distancia de edición.
  - edit-bpm: Algoritmo que explota técnicas de paralelismo a nivel de bit y está optimizado para realizar alineamiento de secuencias relativamente cortas.
- Input: Parámetro que permite definir el fichero de entrada. La pruebas se han realizado con dos ficheros distintos:
  - Dataset1: Fichero que contiene 100 mil pares de secuencias de 1000 caracteres.
  - Dataset2: Fichero que contiene 1 millón de secuencias de 100 caracteres
- Error: Porcentaje que define la tolerancia del filtro. Si dos cadenas presentan un error menor que este valor, el filtro marcará las cadenas para que puedan alinear,

en caso contrario el filtro las marcará como cadenas que no alinean. Las pruebas se han realizado con una tolerancia del 5 %

- kmer-length: Permite elegir la longitud del k-mer.

Dado que hay muchos parámetros, el primer paso será analizar el algoritmo de filtrado modificando la longitud del k-mer y los datos de entrada. El objetivo es detectar cómo varía el rendimiento y la precisión del algoritmo en función de estos parámetros.

5.2. Análisis de rendimiento

La figura 3 muestra el rendimiento obtenido en función de la longitud del k-mer y los datos de entrada para las dos arquitecturas: Intel y ARM. Se puede observar que, independientemente de la distribución de los datos, cuanto mayor es la longitud de la k, peor es el rendimiento. Este deterioro no es proporcional a la longitud del k-mer, ya que para las longitudes mayores que 7 el rendimiento se reduce exponencialmente.
Este comportamiento es debido a que el tamaño del histograma crece exponencialmente en memoria, dado que tiene un tamaño de n^k. Es decir, al ocupar un espacio en memoria exponencialmente más grande, los accesos a memoria son menos eficientes, se producen más fallos en las cachés y se producen mayores penalizaciones por acceso a memoria.

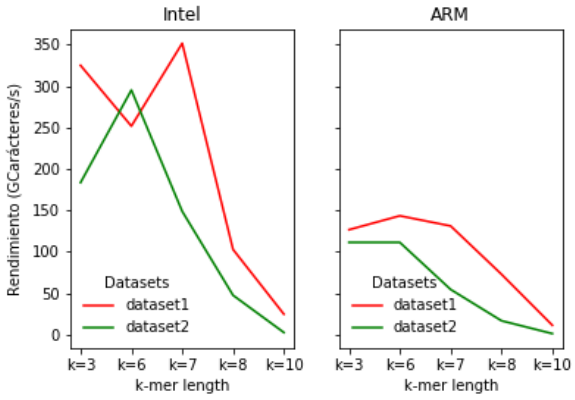

Fig. 3: GCarácteres /s: Rendimiento de la versión base del código ejecutado con distintos datasets y en distintas arquitecturas (ARM e Intel)

Por otro lado, la figura 4 muestra el total de accesos a la cache de último nivel (LLC-loads) y el total de fallos (LLC-loads-mises), para ambos datasets. Se puede observar que a medida que aumenta el tamaño del k-mer, aumentan los accesos a la cache de último nivel de forma exponencial y, de la misma forma, aumentan los fallos.

Aunque los dos datasets procesan la misma cantidad de caracteres, la distribución de estos caracteres afecta de manera considerable al rendimiento del algoritmo. Esto es debido a la existencia de una función de inicialización que se encarga de poner todos los contadores del histograma a zero por cada alineamiento. Por este motivo, se puede observar que el Dataset2 (1M de pares de secuencias) presenta más fallos que el Dataset1 (100K de pares de secuencias). Es decir, el total de inicializaciones que realiza el Dataset2 es 10 veces mayor que el del Dataset1.

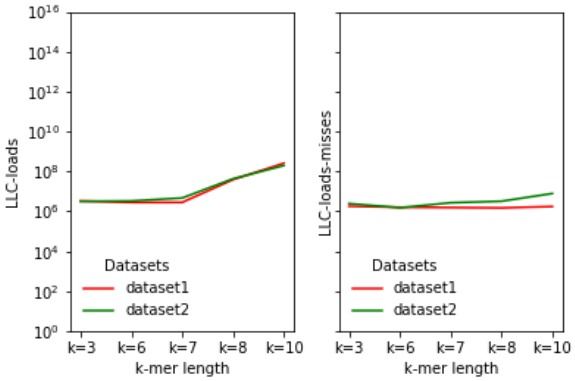

Fig. 4: LLC loads y LLC loads misses obtenidos a partir de la ejecución del código base en las dos arquitecturas (ARM e Intel) con dos datasets distintos.

Por otro lado, se puede observar que el rendimiento en la arquitectura Intel es aproximadamente 2 veces superior al de la arquitectura ARM ($S_{up}(Dataset1, k = 6) = 1,75x$). Para intentar entender cuantitativamente este empeoramiento se ha analizado el IPC. En la figura 5 se puede observar que el IPC se reduce de manera exponencial con la longitud del k-mer. Sin embargo, cabe resaltar que ambas arquitecturas no presentan diferencias considerables con respecto al IPC. Incluso para longitudes de k grandes, el procesador ARM presenta un IPC más alto que el de Intel. Por esta razón, se procede a analizar las instrucciones y ciclos por separado para ver la cantidad exacta ejecutada en cada procesador.

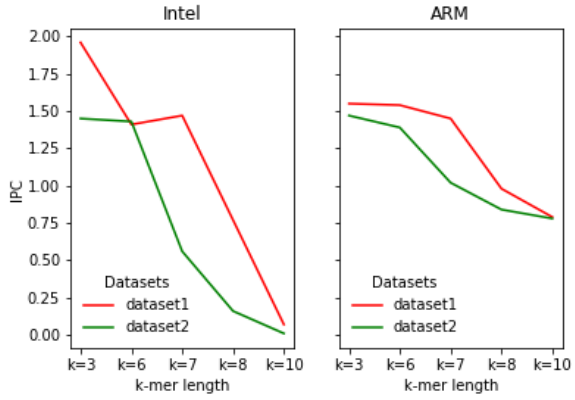

Fig. 5: IPC obtenido a partir de la ejecución del código base en las dos arquitecturas (ARM e Intel) con dos datasets distintos

En la figura 6 se puede observar que las dos arquitecturas presentan prácticamente la misma cantidad de ciclos. No obstante, cuando k es mayor que 7, el procesador ARM ejecuta aproximadamente 1000 instrucciones más que Intel. Por otra parte, cabe destacar que los dos procesadores trabajan con una velocidad de reloj diferente, ya que el procesador Intel trabaja con una frecuencia de $4{,}99GHz$, mientras que ARM trabaja a una frecuencia de $2{,}26GHz$. Es decir, Intel ejecuta el doble de ciclos por segundo que ARM, esta es la principal razón por la que obtiene un speed-up de aproximadamente $2x$.

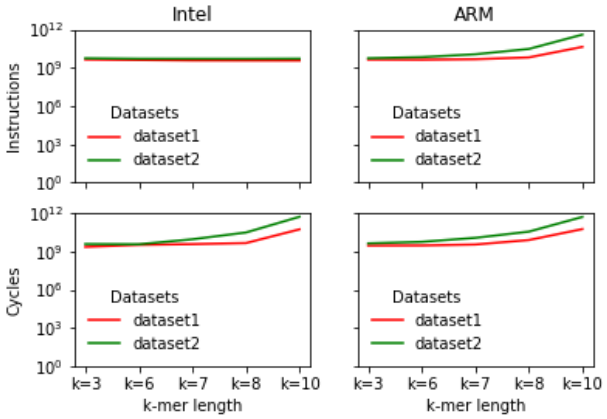

Fig. 6: Instrucciones y ciclos obtenidos a partir de la ejecución del código base en las dos arquitecturas (ARM e Intel) con dos datasets diferentes.

Estos datos indican que el cuello de botella que presenta el algoritmo esta en el acceso a memoria. Aunque el algoritmo de filtrado usa una estructura lineal para realizar el conteo de k-mers, este algoritmo puede llegar a tener un rendimiento peor que otros algoritmos cuadráticos basados en programación dinámica, tal y como se puede observar en la figura 7.

$$S_{up}(Dataset2, k = 12, editBPM) = \frac{80}{0{,}012} = 6666x \quad (2)$$

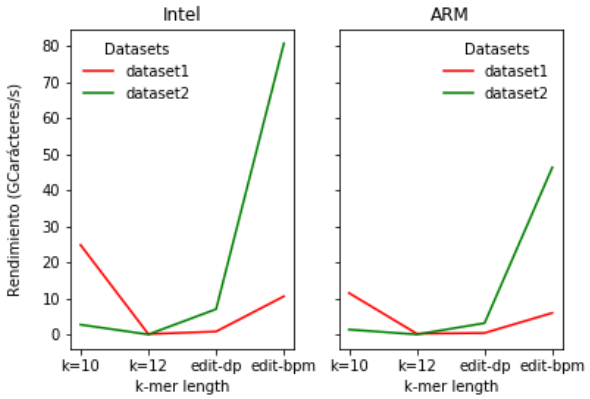

Fig. 7: GCarácteres/s: Comparativa de rendimiento entre k-mer filter y los algoritmos basados en la distancia de edición. Las ejecuciones se han realizado con dos datasets y en las dos arquitecturas (ARM e Intel).

No obstante, el objetivo del proyecto no se basa solo en aumentar el rendimiento del algoritmo, sino que también queremos detectar la longitud óptima del k-mer. La elección de esta longitud representa un balance de varios factores ya que, cuanto más grande sea la k, mayores serán las posibilidades de que un k-mer cubra un error. Es decir, un error en una posición afectará a más k-mers solapados cuanto mayor sea la k. Sin embargo, cuando la longitud es muy pequeña habrá una mayor probabilidad de que ocurran k-mers idénticos de forma espúrea en otras regiones de la secuencia. Por lo tanto, el filtro pierde precisión debido a la gran cantidad de falsos positivos[CM14].
Por este motivo la precisión del algoritmo no aumenta de manera lineal respecto a la longitud del k-mer, ya que existe un punto de inflexión donde el k-mer alcanza la longitud óptima. En la siguiente tabla, se puede observar que, en el caso del Dataset1, la longitud óptima del k-mer es 8. De la misma forma, para el Dataset2, la longitud óptima es 6.

Hits		
k	Dataset1	Dataset2
3	25467	766954
6	97385	956257
7	98193	949652
8	98284	938082
10	97882	901664

6 Optimización de los accesos a memoria

Los resultados del análisis muestran que el cuello de botella es la memoria. El problema principal reside en la inicialización del histograma. Se ha comprobado que cuando la k es mayor que 10, el programa tarda más inicializando el histograma que realizando los propios cálculos del algoritmo. La siguiente figura muestra el pseudocódigo del algoritmo.

Algorithm 1 K-mer Filter

procedure COMPUTE MIN BOUND **Data :** Kmer-length, FileInput
Result: Histogram computed
initialization;
while *i<(MaxSeq/2)* **do**
  $histogram[k] = 0$;
  while $k < n^k$
  do
    Fetch counters and store them in window
    Increment kmer counts: $histogram[k] += 1$;
    Compute min-error bound
  end
end

En la figura 8 se puede observar como varía el rendimiento del algoritmo al quitar la función de inicialización.

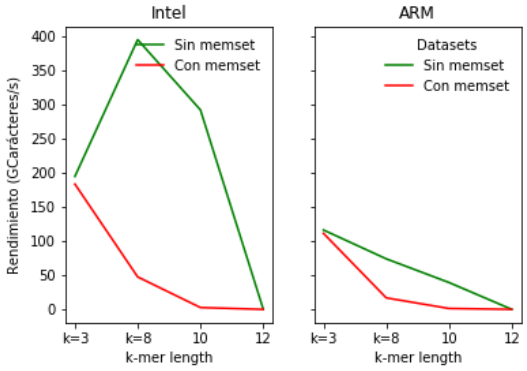

Fig. 8: GCarácteres/s: Rendimiento de dos versiones de código (Con y sin memset) ejecutados con el dataset2 en dos arquitecturas(ARM e Intel).

Para reducir el tiempo de inicialización, se ha decidido cambiar la implementación del código de modo que no se inicializará el histograma entero en todas las ocasiones. Es decir, en cada iteración solo se inicializarán los k-mers que afectan a las cadenas que se estén comparando.
Para ello se identificara cada alineamiento con un «tag» y se creará una estructura unidimensional denominada ”propiedad”que almacenará el tag del último alineamiento que ha utilizado cada contador para todos los elementos del histograma. De esta forma, inicializaremos localmente cada contador según se necesiten atendiendo a el tag propietario del contador. De este modo, el algoritmo queda implementado de la siguiente manera:

Algorithm 2 K-mer Filter

```
procedure COMPUTE MIN BOUND  Data : Kmer-length, FileInput,tag,property
        array, histogram
Result: Histogram computed
while i<(MaxSeq/2) do
    while k < n^k ) do
        if tag! = property[k] then
            property[k]=tag;
            histogram[k] = 0;
        else
            Fetch counters and store them in window;
            Increment kmer counts:histogram[k]+ = 1;
            Compute min-error bound;
        end
    end
end
```

Debido al overhead que produce la estructura ’propiedad’ en la memoria, cuando el tamaño del histograma es pequeño, la versión base incluso puede llegar a presentar un rendimiento más alto que la nueva versión. El gráfico de la figura 3 muestra que el rendimiento empieza a bajar de manera considerable cuando la k alcanza el valor 8 (en el caso el Dataset 1) y el valor 6 (en el caso del Dataset 2). Con estos datos se puede concluir que la nueva versión del código será más eficiente en estos puntos ya que el overhead de la estructura ”propiedad”se vuelve insignificante comparado con el coste de la función de inicialización. En la figura 9 se puede observar como mejora el rendimiento del algoritmo en los puntos mencionados.

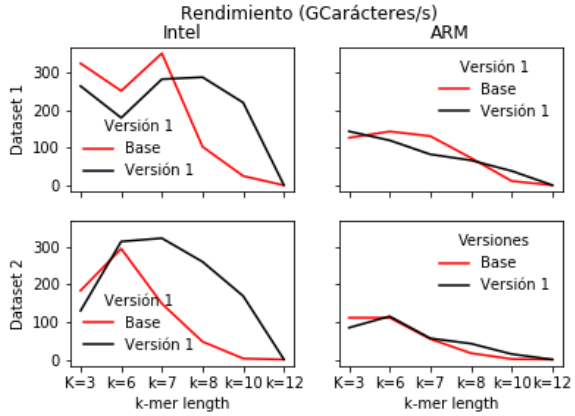

Fig. 9: GCarácteres/s: Rendimiento de las dos versiones de código ejecutados con distintos datasets

Cabe destacar que el rendimiento en el procesador Intel no aumenta de manera proporcional respecto a la longitud del k-mer, ya que aunque se está evitando la inicialización del histograma, se está guardando en memoria la estructura propiedad que también crece exponencialmente. Por este motivo, para k=12 el Speedup disminuye respecto a k=10. En la siguiente tabla se puede observar el speedup obtenido en el procesador Intel.

Ganancia (Speedup) en el procesador Intel		
k	Dataset1	Dataset2
3	0,814034716	0,70729237
6	0,714203514	1,065404096
7	0,80431154	2,170767191
8	2,800737795	5,470317018
10	8,880759823	62,23252195
12	4,794007491	40,62928349

Al tener una estructura más grande no cabe en la memoria cache LLC y por lo tanto se producen más accesos a DRAM, lo que penaliza el rendimiento de manera considerable. Por esta razón en el procesador ARM la versión optimizada no presenta mejoras considerables respecto a la versión original, ya que este posee una memoria cache de 4096K mientras que el procesador Intel posee 16384K. En la figura 10 se puede observar el aumento de los fallos en caché en k=12.

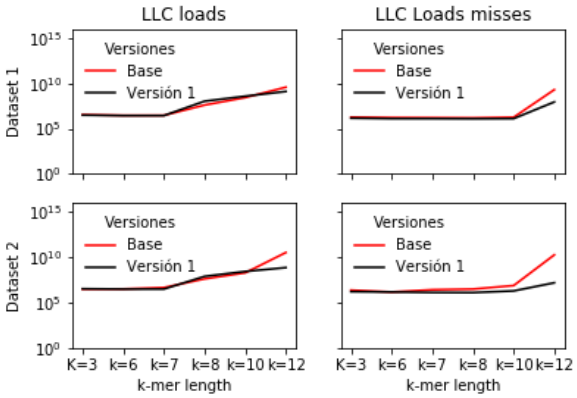

Fig. 10: LLC loads y LLC loads misses obtenidos a partir dela ejecución del código optimizado con los dos datasets.

7 PARALELIZACIÓN DEL CÓDIGO EN CPU

7.1. Introducción

Una vez se han mejorado los accesos en memoria se procederá a paralelizar el algoritmo en CPU mediante OpenMP[Cha+01]. Esta paralelización consiste en repartir las secuencias entre los threads, de manera que cada thread se encargará de computar el histograma de un grupo de secuencias. Debido a la cantidad limitada de threads que ofrece la CPU (en nuestro caso 16), solo es viable aplicar la paralelización de grano grueso. Como el cálculo del histograma de cada secuencia es independiente, los threads pueden manejar los datos de manera paralela tal y como se puede observar en la figura 11.

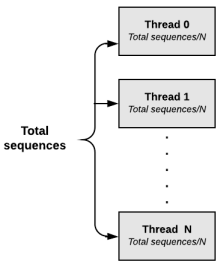

Fig. 11: Distribución de las secuencias entre los threads

7.2. Análisis de rendimiento

La figura 12 muestra el rendimiento obtenido con distintas cantidades de threads. Al aumentar esta cantidad, aumenta el rendimiento. Sin embargo, la ganancia obtenida no es proporcional a los threads que cooperan en computar el trabajo, ya que estos últimos presentan un overhead producido por la creación, destrucción y sincronización. Por otra parte, los resultados extraídos demuestran que la longitud del k-mer afecta al rendimiento de igual manera que la versión secuencial. Esto es debido a que en esta versión estamos aplicando paralelismo de datos. Como se ha explicado en el apartado 5, cuando la longitud del k-mer es mayor que 8 los accesos a memoria se convierten en el cuello de botella.

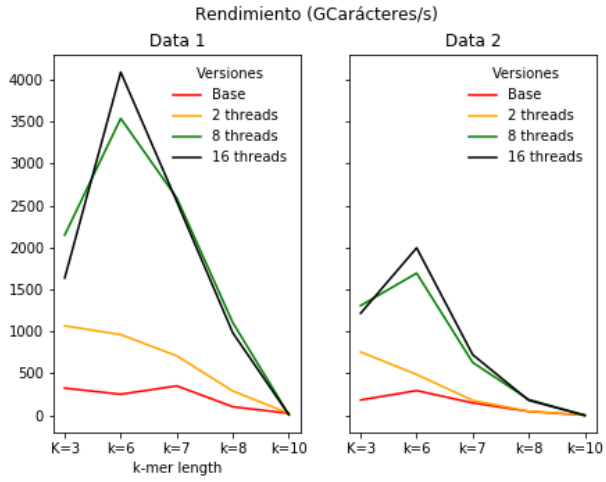

Fig. 12

8 Aceleración del código en GPU

8.1. Introducción

En este apartado se procederá a implementar y analizar una versión acelerada en GPU [LKO05; CPC17] mediante la herramienta Cuda[Nvi]. Para que la integración de la GPU sea óptima, se deben abordar varios problemas. En primer lugar, igual que en la versión implementada con OpenMP, el manejo de los datos debe ser totalmente independiente para poder tratarlos de manera completamente paralela. Por otra parte, igual que en la CPU, se requiere un patrón que minimice los accesos a la memoria global de la GPU.

8.2. Distribución de los datos

Nuestro algoritmo de pre-filtrado presenta una dependencia recursiva para computar los k-mers hallados en una secuencia (Figura 1 y 2). Esta dependencia no ha resultado un problema en la paralelización de CPU, ya que solo se ha aplicado el paralelismo de grano grueso. Sin embargo, debido a la gran cantidad de threads que ofrece la GPU, en esta versión es viable aplicar la paralelización de grano fino.
Para deshacerse de esta dependencia, cada vez que se procederá a computar un k-mer nuevo sin reutilizar cálculos anteriores. Debido a la gran capacidad de cómputo que posee la GPU, este trabajo redundante no afectará en el rendimiento. Una vez eliminada la dependencia, se han implementado las dos versiones.
En la primera versión se ha aplicado el paralelismo de grano grueso (Coarse-grain parallelism), donde cada thread se encarga de calcular el histograma de una o varias secuencias dependiendo del número total de threads contenidos en el Grid. En esta versión cada thread accede a la memoria global para actualizar su histograma.

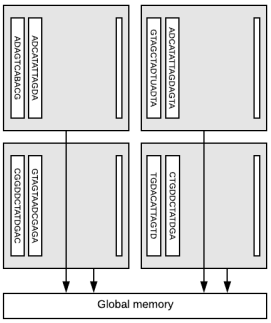

Fig. 13: Paralelismo de grano grueso: Distribución de las secuencias entre los threads hallados en el grid.

La idea de la segunda versión es aprovechar la memoria compartida de cada bloque para reducir los accesos a la memoria global. A diferencia de la versión anterior, en esta versión se ha aplicado el paralelismo de grano fino (Fine-grain parallelism), ya que cada bloque se encarga de computar el histograma de una sola secuencia. De esta forma, cada thread se encargará de realizar el conteo de uno o varios k-mers, dependiendo de la cantidad de threads que posee cada bloque. De este modo, todos los threads actualizarán el histograma de la secuencia en la memoria compartida.

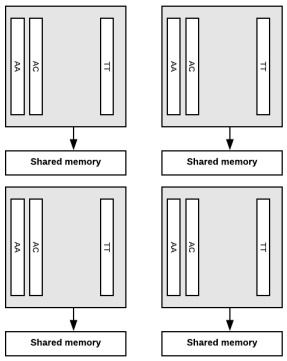

Fig. 14: Paralelismo de grano fino: Distribución de las secuencias entre los bloques hallados en el grid.

8.3. Análisis de rendimiento de la GPU

La figura 15 muestra el rendimiento obtenido de las dos versiones implementadas en la GPU comparadas con la versión base. Dado que los accesos a la memoria global presentan una latencia similar a la DRAM, la paralelización de grano grueso ofrece un rendimiento similar a la versión base. Sin embargo, el uso de la memoria compartida mejora el rendimiento considerablemente presentando un speedup de 8,46 (k=7, dataset 2). Por otra parte, cabe destacar que

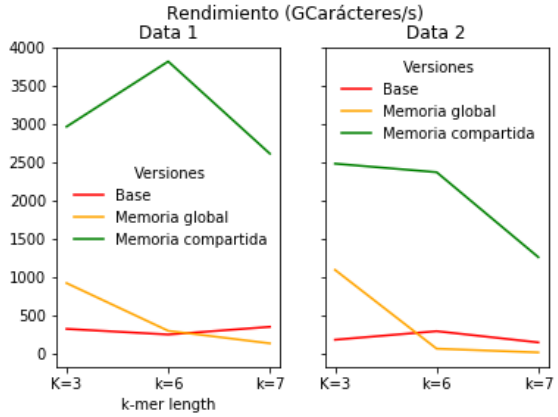

Fig. 15: GCarácteres/s: Rendimiento obtenido de las dos versiones en GPU comparadas con la versión base

para nuestro problema, el paralelismo de grano fino es más viable si la cantidad de secuencias a comparar es grande. Dado que cada thread se encarga de computar uno o varios k-mers, si la cantidad de trabajo a realizar es pequeña, el rendimiento puede empeorar debido al overhead que presentan los threads que computan el histograma de una sola secuencia. En el caso contrario, este overhead se vuelve insignificante comparado con la cantidad de trabajo a realizar. Esto se puede observar en la siguiente figura. La figura 16 muestra el tiempo de ejecución que se tarda en realizar una llamada a la función kernel. Se puede observar que en el dataset 1, la versión que hace uso de la memoria compartida presenta un tiempo de ejecución similar que la versión que hace uso de la memoria global. No obstante, la diferencia de tiempo obtenido con el dataset 2 es considerable.

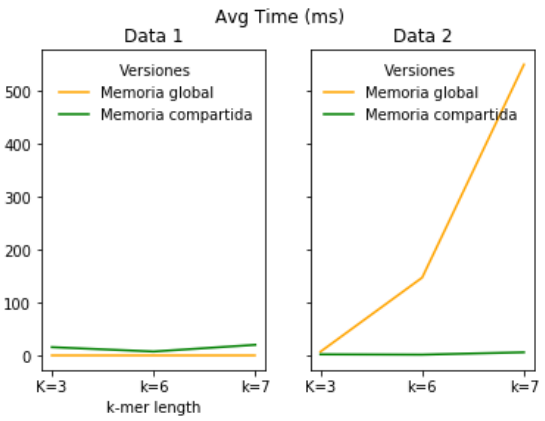

Fig. 16: Tiempo promedio.

Es importante señalar que la memoria compartida de la GPU suele tener un tamaño muy limitado. La memoria compartida de nuestra GPU posee un tamaño de 49,15 KBytes. Dado que el histograma crece exponencialmente en función de la longitud del k-mer, la longitud máxima del k-mer soportada por nuestro programa se limitaría a 7. Para longitudes superiores, como por ejemplo k=8, necesitamos un histograma de 64 KBytes, que traspasa el tamaño que ofrece nuestra memoria.

$$\frac{(4^8)*8bits}{8} = 65536Bytes = 64KBytes \qquad (3)$$

La figura 17 muestra el uso de la memoria compartida en función de la longitud del k-mer.

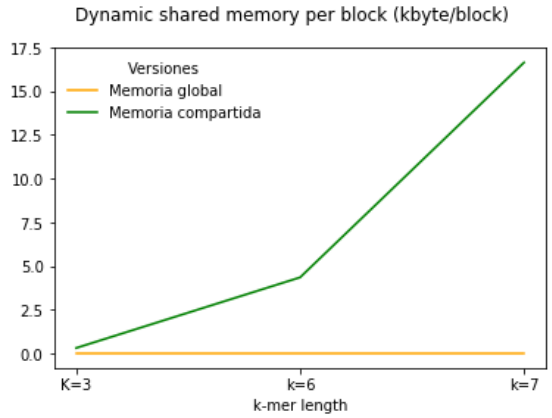

Fig. 17: Uso de la memoria compartida en función de la longitud del k-mer.

9 Conclusiones

En el actual trabajo se ha analizado un algoritmo fundamental en el ámbito de la bioinformática, el filtrado por k-mers, y se ha estudiado su rendimiento con el fin de comprender la importancia que presenta nuestro algoritmo de pre-filtrado. Del mismo modo, se ha realizado un estudio inicial de rendimiento. Aunque nuestra propuesta de algoritmo fue aprovechar la linealidad que presenta para contrarrestar los algoritmos basados en la distancia de edición, se han encontrado varias limitaciones de memoria debido al crecimiento exponencial del histograma que almacena el conteo. Se ha detectado que en algunas ocasiones nuestro algoritmo de pre-filtrado puede presentar peor rendimiento que los algoritmos basados en la distancia de edición. A partir del análisis inicial, se ha implementado una versión para mejorar los accesos a memoria y también se han implementado versiones paralelas. Debido a la gran cantidad de parámetros que requiere el algoritmo, ninguna de las soluciones presentadas en este trabajo ha resultado eficiente en todos los casos. Para nuestro problema, la paralización de grano grueso en GPU no es viable debido a la cantidad de accesos que se producen en la memoria global. La paralelización de grano fino solo es viable si la longitud del k-mer es menor que 7. Para longitudes mayores o igual que 8 resulta más eficiente la versión optimizada. En la figura 18 se puede observar el speedup obtenido en cada versión.

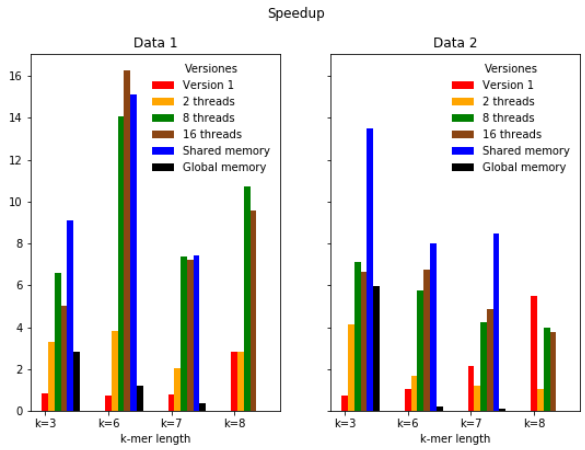

Fig. 18: Speedup obtenido con las distintas versiones implementadas.

10 Agradecimientos

Me gustaría agradecer a Juan Carlos Moure por darme la oportunidad de realizar este trabajo y a Santiago Marco por ayudarme a aprender y progresar con el proyecto en una situación tan difícil como la que estamos viviendo. A nivel personal me gustaría agradecer a mis padres por creer en mi y apoyarme a nivel emocional y económico.

Referencias

[NW70] Saul B Needleman y Christian D Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins". En: *Journal of molecular biology* 48.3 (1970), págs. 443-453.

[SW+81] Temple F Smith, Michael S Waterman y col. "Identification of common molecular subsequences". En: *Journal of molecular biology* 147.1 (1981), págs. 195-197.

[Bur+99] Stefan Burkhardt y col. "q-gram based database searching using a suffix array (QUASAR)". En: *Proceedings of the third annual international conference on Computational molecular biology*. 1999, págs. 77-83.

[Cha+01] Rohit Chandra y col. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[LGG01] Nicholas M Luscombe, Dov Greenbaum y Mark Gerstein. "What is bioinformatics? An introduction and overview". En: *Yearbook of medical informatics* 10.01 (2001), págs. 83-100.

[LKO05] Aaron Lefohn, Joe Kniss y John Owens. "Implementing efficient parallel data structures on GPUs". En: *GPU gems* 2 (2005), págs. 521-545.

[Ryz06] Leonid Ryzhyk. "The ARM Architecture". En: *Chicago University, Illinois, EUA* (2006).

[Cla+10] Francisco Claude y col. "Compressed q-gram indexing for highly repetitive biological sequences". En: *2010 IEEE International Conference on BioInformatics and BioEngineering*. IEEE. 2010, págs. 86-91.

[CM14] Rayan Chikhi y Paul Medvedev. "Informed and automated k-mer size selection for genome assembly". En: *Bioinformatics* 30.1 (2014), págs. 31-37.

[Par15] Mahmoud Parsian. *Data algorithms: Recipes for scaling up with hadoop and spark*. ."Reilly Media, Inc.", 2015, págs. 391-393.

[CPC17] Nicola Cadenelli, Jordà Polo y David Carrera. "Accelerating K-mer frequency counting with GPU and non-volatile memory". En: *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE. 2017, págs. 434-441.

[MS19] Swati C Manekar y Shailesh R Sathe. "Estimating the k-mer Coverage Frequencies in Genomic Datasets: A Comparative Assessment of the State-of-the-art". En: *Current genomics* 20.1 (2019), págs. 2-15.

[Loe] Dr. Laurence Loewe. *Genetic Mutation*. URL: https://www.nature.com/scitable/topicpage/genetic-mutation-1127/.

[Nvi] Nvidia. *CUDA C++ Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[Wik] Wikipedia. *Skylake (microarchitecture)*. URL: https://en.wikipedia.org/wiki/Skylake_(microarchitecture).