Análisis de Suffix-Array y FM-Index en Multi-ManyCore

Alejandro Chacón, Pedro Erencia, Antonio Espinosa, Juan Carlos Moure, Porfidio Hernández[1]

Resumen— **En este artículo se realiza un análisis comparativo del rendimiento entre los algoritmos de búsqueda indexada Suffix-Array y FM-Index actualmente utilizados para acelerar el alineamiento de secuencias de ADN en aplicaciones bioinformáticas. Los dos índices han sido implementados para dos diferentes plataformas paralelas, una CPU Quad-Core y una GPU Nvidia Fermi. Se han implementado diferentes optimizaciones para ambos entornos con el fin de analizar su comportamiento obteniendo mejoras entre 2.3x y 6x dependiendo de la versión. Finalmente los resultados obtenidos revelan que ambos algoritmos de búsqueda están limitados por latencia a memoria y se realizan propuestas de mejora.**

Palabras clave— **Binformática, MultiCore, ManyCore, GPGPU, CUDA, Suffix-Array, FM-Index.**

I. Introducción

El alineamiento de secuencias genómicas es un proceso de gran importancia en el ámbito bioinformático debido a que permite definir las relaciones existentes entre dos secuencias. Esto permite, a su vez, extraer información necesaria para realizar análisis y estudios relacionados con enfermedades hereditarias y estudios filogenéticos, entre muchos otros. Es habitual que estas aplicaciones busquen coincidencias exactas entre fragmentos de secuencias compuestas por cientos de bases y una secuencia de referencia de millones de bases. El coste computacional de este proceso es muy elevado y ha dado lugar a la propuesta y análisis de métodos que utilizan diferentes algoritmos de indexación, como *Suffix-Array* (*SA*)[1] y *Ferragina Manzini Index* (*FM-Index*)[2].

El presente trabajo analiza el rendimiento de varias implementaciones de estos algoritmos sobre sistemas MultiCore y ManyCore. Se utilizará tanto una versión basada directamente en las descripción de la literatura (baseline) como versiones optimizadas para reducir la cantidad de accesos a memoria. Los resultados de la ejecución nos permitirán identificar los problemas de rendimiento de los algoritmos y proponer nuevas estrategias de optimización para futuras líneas de investigación.

El artículo se organiza de la forma siguiente. La siguiente sección explica los algoritmos de indexación *SA* y *FM-Index*. A continuación se introduce la plataforma *CUDA ManyCore*. Después se detallan todas las implementaciones realizadas. En la siguiente sección se muestran los resultados, y finalmente se muestran las conclusiones y el trabajo futuro.

[1]Dpto. de Arquitectura de Computadores y Sistemas operativos, Univ. Autónoma de Barcelona, e-mail: {alejandro.chacon, pedro.erencia, juancarlos.moure, antoniomiguel.espinosa, porfidio.hernandez}@uab.es

II. Indexación de textos

El proceso de alineamiento entre secuencias genéticas se puede traducir a un problema de *matching exacto*.

Dada una secuencia de símbolos $T_{0,n-1}$ considerablemente larga denominada secuencia referencia sobre un alfabeto Σ={1,..,σ} (en nuestro caso {A,C,G,T} de tamaño $\sigma = 4$) y dada una secuencia corta $P_{0,m-1}$ denominada query. El problema del *matching exacto* consiste en encontrar todas las ocurrencias de la query P en la referencia T.

Las técnicas de indexación permiten reducir el tiempo del *matching exacto* a expensas de incrementar notablemente el espacio requerido por la estructura de búsqueda (índice). Los programas de alineamiento más utilizados, como *Bowtie*[3] o *BWA*[4], aceleran este proceso utilizando técnicas de indexación analizadas en el presente trabajo.

A. Índices Suffix-Array

Un índice SA consiste en una permutación de todos los sufijos de T, esta permutación corresponde al orden lexicográfico de los sufijos. La forma más común de representación de un SA es un array con posiciones al texto original T.

Por lo que el SA de un texto $T_{0,n-1}$ es un array $SA[0, n-1]$ que contiene una permutación de posiciones entre el intervalo $[0, n-1]$ tal que el sufijo $T_{SA[i],n-1}$ es menor lexicográficamente que $T_{SA[i+1],n-1}$ para todo $0 \leq i \leq n-1$.

En Fig.1A, se muestra el proceso de creación del SA para el texto $T = acaaacatat$. Para ello se generan todos los sufijos de T y se ordenan lexicográficamente. En Fig.1B se muestra como es posible reducir el espacio de la estructura SA utilizando apuntadores a los sufijos de T.

A.1 Proceso de búsqueda Suffix-Array

El algoritmo de búsqueda retorna el intervalo de posiciones (L y R) del SA donde la query P hace *matching exacto* con los sufijos de T. El algoritmo utiliza la búsqueda dicotómica para acotar el intervalo a retornar. En cada iteración realiza una comparación entre la query P y el sufijo central $M = T_{SA[\frac{R+L}{2}]}$ del intervalo actual L y R. El resultado de la comparación permite acotar en cada iteración el intervalo de búsqueda, actualizando L o R con la posición de M. Por lo que el algoritmo de búsqueda en el SA tiene una complejidad temporal $O(m \times \log_2 n)$.

Debido a que la búsqueda dicotómica estrecha el espacio de búsqueda, llega un punto en que todos los

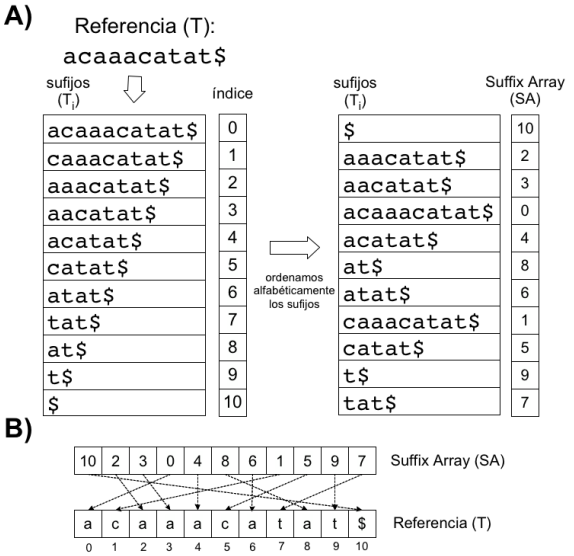

Fig. 1. A) Proceso de generación del índice SA a partir del texto $T = acaaacatat$. B) Substitución de los sufijos por apuntadores al texto original T.

sufijos entre L y R comparten prefijo. Este prefijo es denominado *longest common prefix* (lcp)[1], es posible obviar este prefijo para reducir la complejidad a $O(m + \log_2 n)$.

Es posible que el SA contenga varios sufijos que comparten un prefijo (lcp) de tamaño m, es decir si algún $T_{SA[i],SA[i+m]} == T_{SA[j],SA[j+m]}$ para $i \neq j$. En ese caso, es necesario realizar dos búsquedas, una para acotar L y otra para R.

La parte superior de la Fig.2 muestra el proceso de búsqueda para una query $P = acaaa$ utilizando el SA de $T = acaaacatat$. Como los punteros L y R acotan el espacio de búsqueda hasta encontrar las posiciones con el sufijo buscado.

La complejidad espacial de la estructura SA es $O(n \times \log_2 n)$ y al ser necesario T la complejidad aumenta a $((n \times \log_2 n) + (n \times H_0(T)))$bits siendo $n \times H_0(T)$ la entropia de orden 0 de T. En la práctica, la mayoría de implementaciones utilizan un requisito en memoria de $4n$ Bytes del SA más n Bytes de T con el fin de simplificar la implementación.

B. Índices Ferragina Manzini

Los *self-index* son índices que no precisan del texto original para realizar las búsquedas. Permiten una reducción del tamaño del índice al no incluir T. Una estructura que hace uso de esta técnica es el índice *FM-Index*. Una de las grandes virtudes del *FM-Index* es que permite almacenar el índice en un menor espacio que el SA. Para ello, hace uso de la transformada del texto *Burrows-Wheeler* (BWT)[5]. BWT transforma el texto T en una representación alternativa denotada como T^{bwt}. Esta transformación a T^{bwt} es reversible y es posible recuperar el texto original T en cualquier momento.

La Fig.3 muestra en proceso de generación del BWT. Las rotaciones de todos los sufijos de $T = acaaacatat\$$ y su posterior ordenación lexicográfica para obtener la representación $T^{bwt} = tca\$atcaaaa$ en la última columna de la matriz M.

Para recuperar el texto original se utiliza una propiedad de la transformada denominada *LF-Mapping*[5]. *LF-Mapping* realiza una traslación entre la primera columna (F) y la última columna (L) de la matriz M. F es una permutación de los sufijos de T ordenados lexicograficamente, se puede decir que F representa el SA, mientras que L será nuestra transformada T^{bwt}. Esta relación entre L y F permite obtener la posición en el SA del sufijo anterior al actual. Esto es, la posición del sufijo LF-Mapping$(T_{SA[i],n-1}) = T_{SA[i]-1,n-1}$. En la Fig.3 el sufijo $T = tat\$$ de la posición 8 del SA es el sufijo anterior al sufijo $T = at\$$ que se encuentra en la posición 5, por tanto LF-Mapping(5)=8.

Debido a esto si se realiza *LF-Mapping* $|T|$ veces desde el carácter \$ del T^{bwt}, se obtiene el texto original T desde el final al principio.

Dado que T y T^{bwt} tienen la misma longitud n, la complejidad espacial de la estructura *FM-Index* es O(n). Donde es posible representar la información del texto T^{bwt} en $n \times H_0(T^{bwt})$bits.

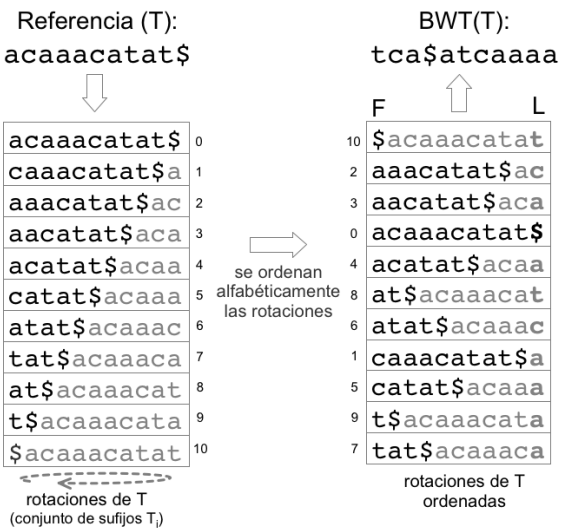

Fig. 3. Proceso de generación de la transformada BWT del texto $T = acaacatat$.

B.1 Proceso de búsqueda FM-Index

Los *backward-search index*[6] tienen la característica de realizar la búsqueda de la query P carácter a carácter en orden inverso, de P_{m-1} a P_0. *FM-Index* pertenece a este tipo de índices y aprovecha la propiedad del *LF-Mapping* para poder realizar búsquedas indexadas en el texto.

El proceso de búsqueda consta de m pasos, 2 apuntadores L y R que actualizan su posición utilizando la propiedad *LF-Mapping*, permitiendo acotar el número de sufijos de la columna F como resultado de la búsqueda. *LF-Mapping* se aplica realizando dos operaciones de conteo ($C + Occ$) sobre los caracteres de T^{bwt}. C corresponde al número total de caracteres en T^{bwt} lexicográficamente inferiores a P_i, siendo P_i un carácter de P. Occ es el número de caracteres P_i que hay por delante

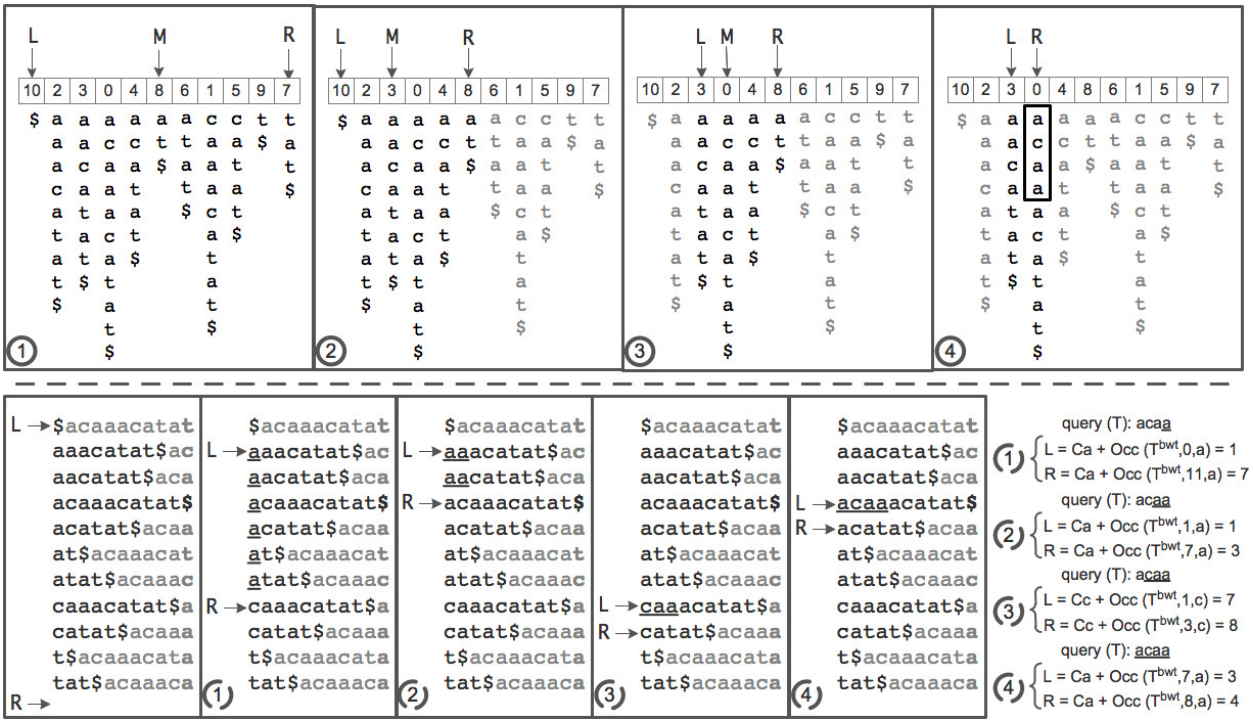

Fig. 2. Proceso de búsqueda de la query $P = acaa$ en el texto $T = acaaacata$. En la parte superior la búsqueda se realiza sobre el SA y en la parte inferior sobre el FM-Index.

del apuntador actual (L_j o R_j) en el T^{bwt}. Por tanto, la siguiente posición de los apuntadores L y R se calcula como $\{L_{j+1}, R_{j+1}\} = C(T^{bwt}, P_i) + Occ(T^{bwt}, \{L_j, R_j\}, P_i)$ para todo $m - 1 \geq i \geq 0$ y $0 \leq j \leq m - 1$.

En la parte inferior de Fig.2 se muestra el proceso de búsqueda de $P = acaa$ en $T = acaaacatat\$$ utilizando el *FM-Index*. Donde L_j y R_j acotan el resultado a cada paso.

La complejidad temporal de la búsqueda del *FM-Index* es O(m) y depende únicamente del tamaño de la query P. En la práctica, en el proceso de generación del *FM-Index* cada cierto intervalo fijo de caracteres se añaden unos contadores pre-computados para acelerar el cálculo de C y Occ, esto incrementa el tamaño del índice.

III. Compute Unified Device Architecture

En el 2007 *Nvidia* lanzó un entorno de herramientas y compiladores denominado *Compute Unified Device Architecture* (CUDA)[7], que permiten desarrollar y ejecutar aplicaciones de cómputo en sus dispositivos gráficos. En este trabajo CUDA es utilizado para realizar y evaluar un seguido de implementaciones de los algoritmos SA y *FM-Index* en arquitecturas GPU.

Los procesadores Multicore perciben la GPU como un co-procesador al que mandar tanto los datos como las tareas de cómputo, denominadas kernels. Los cuales deben tener un paralelismo masivo subyacente para ser eficientes.

A. Arquitectura Nvidia Fermi

Fermi es una arquitectura ManyCore de *Nvidia*. Esta arquitectura permite un cómputo y ancho de banda un orden de magnitud mayor a los ofrecidos por arquitecturas MultiCore. Ejecuta decenas de miles de threads simultáneamente, necesarios para explotar eficientemente los recursos.

Fermi se compone de un conjunto de unidades funcionales independientes llamadas *Stream Multiprocessors* (SM) las cuales comparten una *caché L2* de 756KB. Cada uno de los SM contiene: 32 *Stream Processors* (SP), 16 recursos de Load/Store y 4 unidades *Special Function Unit* (SFU) para operaciones transcendentales. Todos estos recursos comparten un *File Register* con 32K registros, una *cache L1* y *Shared Memory* programable a 48KB o 16KB.

Cada SM contiene 2 planificadores que pueden lanzar instrucciones simultáneamente. Cada instrucción lanzada debe ser ejecutada simultáneamente por 32 threads distintos, a esta agrupación se le denomina *warp*. A este paradigma de ejecución Nvidia le denomina *Single Instruction Multiple Thread* (SIMT), y tiene una implicación directa en el rendimiento.

Todos los threads tienen acceso a una memoria de almacenamiento global de la propia GPU y puede ofrecer un ancho de banda de hasta 190GB/s. Para explotar todo el ancho de banda es necesario que los threads realicen accesos a posiciones contiguas en memoria (*coalesced*). La memoria de almacenamiento global de la GPU es GDDR5 y su capacidad máxima es actualmente de 6GB. Esta limitación de memoria puede ser un problema añadido al trabajar con estructuras de datos muy grandes, como por ejemplo los textos indexados.

IV. Implementaciones

En el actual capítulo del artículo se describen las implementaciones realizadas de la búsqueda para los índices *FM-Index* y SA para entornos paralelos MultiCore y ManyCore. Para cada índice se implementa una versión denominada BaseLine que es una propuesta naïve y otra con un conjunto de optimizaciones.

Para todas las implementaciones se ha utilizado la misma estrategia de paralelización. Tenemos un conjunto de queries P a buscar sus ocurrencias, cada una de estas búsquedas es independiente y pueden realizarse de forma simultánea. En la versión MultiCore cada thread se reparte equitativamente el conjunto de queries P, mientras que en GPU se genera un thread por cada query P.

A. Implementación Naïve SA

Esta primera implementación del SA utiliza dos estructuras, un array de enteros para almacenar el SA con un tamaño de 4n Bytes y la referencia T como un array de caracteres ASCII que ocupa n Bytes. Se realiza una doble búsqueda dicotómica, como se explica en el capítulo II-A, para encontrar las ocurrencias de P. La primera para acotar L y después otra para acotar R.

La búsqueda compara P con $log_2 n$ sufijos de T. Cada una de estas comparaciones precisa como mínimo de 3 accesos. Un acceso a P, un acceso al SA y un segundo acceso encadenado a T. Este último acceso es dependiente de los datos del SA, por lo que es no es predecible. El cálculo de la comparación se realiza base a base, así pues, si un carácter no fuera suficiente para calcular el resultado de la comparación, entonces, es necesario nuevamente acceder y comparar el siguiente carácter de T y P.

Esta versión evita realizar cómputo redundante en las comparaciones al no comparar la parte del *lcp* común a los sufijos que se encuentran entre L y R, ver capítulo II-A.

B. Optimizaciones SA

Con el fin de reducir el requisito de almacenamiento en memoria se ha compactado la referencia T en tiempo de generación del índice. La compactación se realiza modificado la representación de la referencia T, utilizando 2 bits por carácter, esto es $n \times \log_2 \sigma$ bits para almacenar T. Lo que supone reducir a 1/4 el requisito en memoria de T.

Las queries también son compactadas de la misma forma, con el fin de facilitar la comparación entre queries y sufijos. Este proceso se realiza en tiempo de ejecución antes de realizar la búsqueda.

Para reducir el número de comparaciones y accesos a memoria, la compactación de T y P se almacena en forma de bitmaps de 4 Bytes (16 caracteres) denominados *bitmapT* y *bitmapP*. El número de accesos que realiza el algoritmo se reduce, debido a que cada vez que se accede al sufijo de T comparamos los 16 caracteres de forma simultánea.

Esta reducción de accesos a memoria conlleva un incremento en las operaciones de cómputo al realizar un alineamiento del sufijo T con la query P para cada comparación. El proceso de alineamiento implica concatenar un sufijo de $bitmapT_i$ con un prefijo de $bitmapT_{i+1}$ para generar el bitmap alineado a comparar con $bitmapP_j$.

Esta implementación también evita realizar cómputo redundante al no analizar el *lcp* común entre L y R, pero en esta versión debido a las optimizaciones se evita cada 16 caracteres de P.

C. Implementación Naïve FM-Index

Esta implementación del *FM-Index* utiliza los contadores pre-computados comentados en el capítulo II-B. Cada 64 caracteres de T^{bwt} se almacenan σ contadores (un contador para cada base {A,C,G,T}) en un array especifico para ello. Al ser cada elemento un entero, esta estructura ocupa $n/4$ Bytes. Por tanto, el requisito en memoria del índice (T^{bwt} + contadores) es $n/4 + n$ Bytes.

De esta forma para aplicar *LF-Mapping* no es necesario realizar una operación de conteo sobre todos los caracteres del T^{bwt} ya que cada contador almacena a partir de su posición las ocurrencias de la operación $C + Occ$. Por lo tanto, en el proceso de búsqueda únicamente es necesario contar los caracteres posicionados entre los contadores, que en este caso como máximo implica verificar 63 caracteres.

Calcular el *LF-Mapping* supone como mínimo 2 accesos a memoria, un acceso a un carácter de P y un acceso a la estructura de los contadores pre-computados. Además, el cálculo del Occ puede suponer un incremento en el número de accesos entre 0 y 63 dependiendo de la posición del apuntador (L o R) al T^{bwt}.

Se puede apreciar que incrementar el intervalo de los contadores reduce el requisito en memoria del índice pero por contra aumenta el cómputo a realizar por Occ.

D. Optimizaciones FM-Index

Las optimizaciones de esta implementación están orientadas a reducir el cómputo y el número de accesos a memoria en la búsqueda.

Para ello se ha utilizado la instrucción *popcount* incluida en el *ISA* de las arquitecturas utilizadas (*nehalem* y *fermi*). Esta instrucción permite contar el número de bits a 1 que contiene un registro.

Para hacer uso de ella se ha necesitado compactar el T^{bwt} en bitmaps de 64 bits. Cada 64 caracteres corresponden a 2 bitmaps. Esto permite reducir el cálculo de $C + Occ$ a un seguido de operaciones lógicas y una operación *popcount* independientemente de la posición del apuntador en T^{bwt}. Además se agrupa cada pareja de contadores y bitmaps en posiciones consecutivas en el índice. De este modo se puede acceder a ambos valores con un único acceso.

Por tanto, el número mínimo de accesos a memoria necesarios para realizar *LF-Mapping* queda reducido a 2, uno para leer un carácter de P y un segundo

acceso para leer el contador y los dos bitmaps de 64bits.

V. Resultados

En este capítulo se detalla el entorno utilizado, la instrumentación de las implementaciones detalladas en el capítulo anterior y los resultados obtenidos de estas mediciones.

A. Entorno y metodología de experimentación

El entorno de experimentación consta de una máquina con las siguientes caracteristicas: Sistema Operativo *Debian 6.0*. Un procesador *Intel core i7 920* de 4 núcleos con *Hyperthreading* a 2,6Ghz con arquitectura *nehalem*, con un ancho de banda de 25GB/s y 8GB de memoria *DDR3*, con *TurboBoost* a 2,93Ghz. Una GPU *Nvidia GTX 480* con 480 cores a 1,4Ghz con arquitectura *fermi* y equipada con 1,5GB de memoria *GDDR5* con un ancho de banda de 177GB/s. Se ha utilizado *GCC v4.4.5* y el *toolkit CUDA 4.1* para la compilación y se han compilado explícitamente para arquitectura de 64 bits, *compute capability* 2.0 y optimizaciones O3. En las mediciones se ha realizado el promedio de 5 ejecuciones para obtener los tiempos. Midiendo siempre únicamente la parte de la búsqueda de los algoritmos. Por lo que las mediciones de GPU no se tiene en cuenta el tiempo en transacciones de datos entre CPU y GPU.

Debido al *TurboBoost* la frecuencia del procesador es variable dependiendo del número de núcleos activos. En el caso serie el procesador se ejecuta a 2,93Ghz, y con 4 u 8 threads activos el procesador se ejecuta a 2,6Ghz. Teniendo en cuenta esta variación de frecuencia el speedup ideal alcanzable del procesador para 4 threads es 3,56 y con 8 threads de 7,12.

Como workload de entrada para ejecutar las búsquedas se ha utilizado el *Genoma Humano* completo ($\approx 3GB$) que se encuentra en la base de datos de libre acceso *ncbi* al cual se le han extraído las bases N. Las ejecuciones se realizan utilizando prefijos de esta referencia de diversos tamaños. Las mediciones se realizan con 1 millón de queries aleatorias de 100 bases, diferentes para cada ejecución. Para la generación de estas queries se utiliza una herramienta del proyecto *MUMmer*[9] que es ampliamente utilizada en este tipo de pruebas. Esta herramienta extrae queries de una referencia, en nuestro caso el *Genoma Humano*, introduciendo mutaciones con una cierta aleatoriedad.

Todos los resultados de los algoritmos se han verificado con una implementación de control. Esta implementación se puede encontrar en *Pizza&Chili Corpus*[8] y esta validada por la comunidad.

B. Resultados

Las Fig. 4 y 5 muestran el tiempo en procesar un conjunto de queries variando el tamaño de la referencia, para las implementaciones BaseLine (BL) y Optimizada (OPT), y para el *SA* y *FM-Index*. Las ejecuciones en CPU son ejecutadas para 8 threads y en GPU para bloques de 256 threads. Cada uno de estos 3 factores será analizado por separado.

Para el índice *SA* no es posible obtener tiempos para una referencia mayor a 200Mbases ya que el tamaño del índice resultante sobrepasa la memoria principal disponible en la GPU.

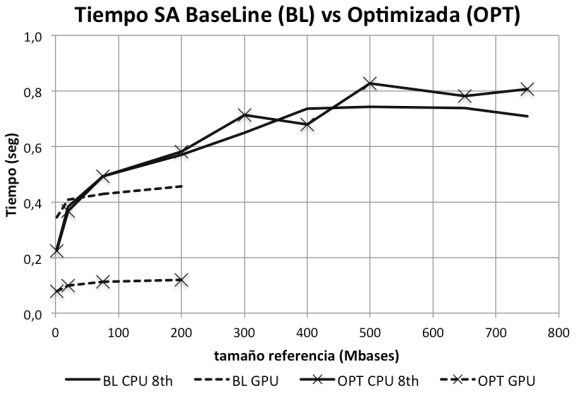

Fig. 4. Tiempos de la versión BaseLine y optimizada SA.

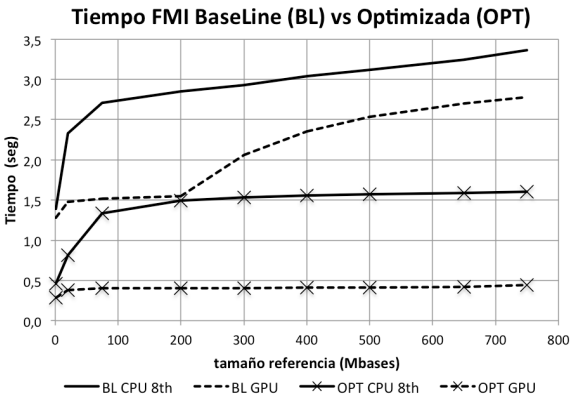

Fig. 5. Tiempos de la versión BaseLine y optimizada FMI.

B.1 Versión BaseLine frente a versión Optimizada

Podemos observar en Fig. 4 que la versión optimizada en GPU obtiene una ganancia de $\approx 4\times$ frente a versión no optimizada, en cambio, en CPU su rendimiento se degrada ($\approx 0.9\times$). El cambio de accesos a memoria por cómputo de OPT beneficia a la GPU por contra la CPU se encuentra penalizada por este cómputo adicional debido a que el cómputo añadido se encuentra en el camino crítico.

Con el *FM-Index* (Fig. 5) la reducción del número de accesos al índice y el uso de instrucciones *popcount* específicas de la arquitectura se ven reflejadas con un aumento del rendimiento: $\approx 2.3\times$ en la CPU y $\approx 6\times$ en la GPU.

B.2 CPU frente GPU

Todas las versiones GPU mejoran el rendimiento frente a las versiones CPU para referencias de tamaños razonables. La mayor ganancia se aprecia con SA OPT ($\approx 6\times$). Esta mejora realmente es debida a que la versión CPU OPT incluso empeora su rendimiento. Por otro lado, para la versión BL la

ganancia es de $\approx 1.3\times$. Para el *FM-Index* la ganancia es de $\approx 4\times$ en la versión OPT y de $\approx 1,3\times$ para la versión BL.

Podemos observar que este tipo de algoritmos tienen un mayor rendimiento en GPU, posiblemente debido a la capacidad de las GPUs de ocultar la latencia en memoria de la búsqueda de varias queries simultáneas.

B.3 Suffix-Array frente FM-Index

La Fig. 6 muestra que todos los tiempos de ejecución del SA son entre $\approx 2\times - 4\times$ mejores que con *FM-Index*. Así pues, mientras que el SA permite un mayor rendimiento, el *FM-Index* tiene un tamaño más reducido y permite ejecutar mayores tamaños de referencia en GPU.

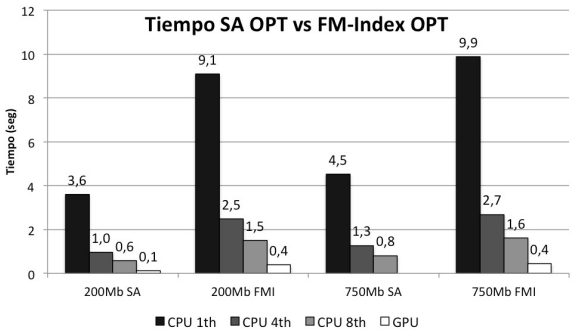

Fig. 6. Tiempos de la versión optimizada del SA frente al FMI.

B.4 Escalabilidad en CPU

La tabla I permite analizar la escalabilidad que presentan cada una de las implementaciones en CPU. Para ello se muestra el speedup de cada implementación frente a su versión serie para un tamaño de referencia fijado en 200Mbases. Los datos muestran que para 4 threads todas las aplicaciones escalan en la CPU y se alcanza el speedup ideal de 3.56 (descrito en el capítulo V) y con 8 threads se alcanza entre el 80 y 86% del speedup ideal.

TABLA I
ESCALABILIDAD DEL SA Y FM-INDEX PARA TAMAÑO DE REFERENCIA DE 200MBASES EN MULTI-MANYCORE

	Suffix-Array		FM-Index		
SP	BL	OPT	BL	OPT	Efic.
4th	3.6x	3.5x	3.6x	3.6x	100%
8th	5.7x	5.6x	5.5x	6.1x	80-86%
GPU	7.4x	31.8x	10.4x	22.3x	

Observando el speedup que describen todas las implementaciones CPU con 8 threads utilizando hyperthreading. Podemos decir que la mejora obtenida del 80-86% frente a la versión de 4 threads es debida a la baja utilización de la capacidad de los recursos del core. Los threads no están aprovechando los recursos del core, de lo que se puede extraer que los algoritmos de búsqueda presentan un problema de latencias.

Como se ha podido observar el uso del multithreading permite ocultar estas latencias y mejorar la utilización de la capacidad de los recursos. La estrategia de procesar la búsqueda de varias queries de forma simultánea beneficia a la ejecución. No debemos obviar que en CPU se realizan 2 búsquedas de queries por core de forma simultánea frente a las 1536 queries por SM que se buscan en la GPU.

VI. CONCLUSIONES

La evaluación experimental muestra una mejora del rendimiento de los algoritmos asociada al uso de la característica multi-threading de los procesadores. Además, el aumento del rendimiento al implementar optimizaciones que reducen los accesos a memoria en detrimento de aumentar el cómputo, muestran que un problema de los algoritmos de búsqueda son las latencias a memoria.

Se realizará un análisis más detallado en el cual se midan contadores hardware relacionados con fallos de caché, accesos a memoria y ancho de banda consumido. Este análisis es necesario para identificar las dependencias causantes de las latencias y verificar el margen de mejora en CPU y GPU para estos algoritmos.

Para CPU se propone implementar y evaluar una versión que simule el multi-threading por software realizando cada thread diferentes búsquedas simultáneas. Y para GPU realizar optimizaciones orientadas a reducir el número de accesos a memoria y explotar el ancho de banda disponible. Para lograr alcanzar esa mejora en el rendimiento.

AGRADECIMIENTOS

El presente trabajo ha sido financiado mediante el proyecto MICINN-Spain TIN2011-28689-C02-01.

REFERENCIAS

[1] U. Manber G. Myers, *Suffix arrays: a new method for online string searches*, SIAM Journal on Computing, Volume 22, Issue 5, 1993 P. Ferragina, G. Manzini,

[2] *Opportunistic Data Structures with Applications*, Proceedings of the 41st Annual Symposium on Foundations of Computer Science. p.390, 2000

[3] B. Langmead, C. Trapnell, M. Pop, SL. Salzberg, *Ultrafast and memory-efficient alignment of short DNA sequences to the human genome*, Genome Biology 10:R25, 2009.

[4] H. Li, R. Durbin, (2009) Fast and accurate short read alignment with Burrows-Wheeler Transform. Bioinformatics, 25:1754-60. [PMID: 19451168],

[5] M Burrows, D Wheeler, *A block sorting lossless data compression algorithm*, Technical Report 124, Digital Equipment Corporation, 1994

[6] G Navarro, V Mäkinen, *Compressed full-text indexes*, ACM Computing Surveys (CSUR) 39 (1), 2, 2007

[7] D. Kirk and W. Hwu, *Programming Massively Parallel Processors*, Morgan Kaufmann, ISBN 978-0-12-381472-2, 2010

[8] P. Ferragina, R. González, G. Navarro and R. Venturini, *Compressed text indexes: From theory to practice*, Journal of Experimental Algorithmics (JEA) Volume 13, Article No. 12 , 2009

[9] S. Kurtz, A. Phillippy, A.L. Delcher, M. Smoot, M. Shumway, C. Antonescu, S.L. Salzberg, *Versatile and open software for comparing large genomes*, Genome Biology 5:R12, 2004.