Optimization of BLAST seed indexing in the alignment of DNA sequences with GPU using CUDA

1st Franklin L. A. Cruz-Gamero
Ciencia de la Computación
Universidad Nacional de San Agustín
Arequipa, Perú
fcruz@unsa.edu.pe

2nd Juan C. Gutiérrez-Cáceres
Ciencia de la Computación
Universidad Nacional de San Agustín
Arequipa, Perú
jgutierrezca@unsa.edu.pe

***Abstract*—In the alignment of biological sequences such as DNA, RNA and proteins, different algorithms are used, mainly the Basic Local Alignment Search Tool (BLAST), which has two phases, a heuristic phase of seed indexing and another extension phase with a comparison of sequences using the Smith-Waterman (SW) algorithm, which allows the alignment of a short sequence "*query*" with a long reference sequence "*db*" in a very fast way in relation to other algorithms of alignment. This work proposes to use a two-dimensional matrix instead of a sparse matrix as a hash table for the storage of the *seed index* obtained, as well as the use of the GPU of our graphic card to optimize the planting, it reduces 11.24 % of the time of processing of seed indexing phase of the BLAST, presenting the use of GPU with CUDA a better performance in processing time than the sequential implementation and another multi CPUs using threads with OPENMP. Our algorithm has a complexity in time of $O(1)$ to obtain the seeds identical to the pattern key. The performance is greater when the length of the hash key increases. For its evaluation tests we used a laptop core i7 of 16gb of RAM and a graphic card of 384 cores with C++ programming language and CUDA. Alignment tests were performed using real DNA sequences obtained from the National Center for Biotechnology Information (NCBI) and ENSEMBL in FASTA format with reference sequences of up to 1.3 Gb, such as the complete genome of the hen (*Gallus gallus*) that has 1 230 258 557 base pairs (bp) and with a query sequence of 140 bp, which was indexed with a 5 bp key in 1074 milliseconds using GPU.**

The source code is available at https://bitbucket.org/FranklinAQP/alineamiento-secuencias-dna/src

***Index Terms*—alignment, GPU, BLAST, optimization, CUDA, DNA**

I. Introducción

El secuenciamiento de un genoma es el proceso de obtener la secuencia completa de ADN de una especie, de igual modo se puede obtener la secuencia de un fragmento de este genoma (cromosoma, gen, etc.). Existen dos métodos de hacerlo:

- La primera es reconstruir el genoma completamente desde cero o "*de novo*".
- La segunda es reconstruir el genoma tomando como referencia otro genoma de la misma especie.

El genoma humano secuenciado posee alrededor de 3 200 millones de pares de bases (pb). Requiriendo para su secuenciamiento por el segundo método, el uso de algoritmos de alineamiento de secuencias de ADN. Donde las secuencias a alinear o de consulta "**query**" son de corta longitud, siendo de 25 a 35 pb en *Illumina* y *SOLiD* [1].

Los algoritmos de alineamiento se pueden clasificar en cuatro grupos por sus características: basados en árboles de sufijos, basados en Tablas Hash, relacionados a Merge Sorting [2] y relacionados con algoritmos evolutivos [3]. Siendo uno de los más usados el algoritmo *Basic Local Aligment Search Tool* o BLAST [4], el cual está basado en Tablas Hash.

El *National Center for Biotechnology Information* (NCBI) ha implementado una herramienta basada en consultas para el alineamiento de secuencias de ácidos nucleicos y aminoácidos que tiene el mismo nombre BLAST que el algoritmo de alineamiento de secuencias BLAST [4]. Esta herramienta utiliza análisis de biosecuencias utilizando implementaciones de perfil oculto de Markov [5] y posee sub-herramientas como BLASTN, BLASTP y BLASTX que procesan alineamientos de nucleótidos, alineamiento de proteínas y la traducción de ADN a proteínas respectivamente.

La aparición de clusters de computación de alto rendimiento, programación distribuida y unidades de procesamiento gráfico (GPU) con mayor número de núcleos cada vez, proporciona una programabilidad más fácil y escalable, lo que permite a los supercomputadores procesar múltiples consultas de manera eficiente y rápida, administrando gran cantidad de datos. Así una pC puede usar su GPU para acelerar la ejecución de muchos procesos al poseer memoria independiente, ancho de banda y alto potencial computacional, donde la clave del rendimiento en esta plataforma es usar multithreading masivo para usar la gran cantidad de núcleos y ocultar la latencia de la memoria

global [6].

El algoritmo de BLAST posee dos fases en base al paradigma "***seed and extend***", siendo la primera fase, la obtención de un índice de semillas o "*seed index*" de la secuencia de referencia en una Tabla Hash que es la agrupación de punteros a secuencias que tienen una misma secuencia de inicio o "*K-mer sequence*" como clave Hash y la segunda fase es la extensión de todas las semillas que coinciden con la semilla de la secuencia de consulta "*query*" usando para su alineamiento el algoritmo de Smith-Waterman [7].

II. Estado del Arte

El algoritmo BLAST posee varias versiones paralelizadas desde su versión inicial [4], cuya característica común es el uso de un entorno distribuido como: GridBLAST [8] usando un *grid computing framework*, DC-BLAST [9] optimizando el tiempo de las búsquedas en la plataforma NCBI BLAST al distribuir las secuencias de consulta, CloudBLAST [10] combina *MapReduce* y Virtualización con el uso de varios nodos, mpiBLAST [11], [12] usando *Message Passing Interface* MPI, HPC-BLAST [13] implementando BLAST con *High Performance Computing* HPC en clusters Intel Xeon Phi.

Otras versiones paralelizadas incluyen además de un entorno distribuido, el uso de GPU pero enfocado en la segunda fase del algoritmo BLAST que es la búsqueda de secuencias similares o "***Smith-Waterman***" como: GPU BLAST [14] usando CUDA con OPENMP y MPI exclusivamente en proteínas para acelerar el algoritmo BLAST enfocado en la paralelización con GPU en la segunda fase Smith-Waterman, G-BLASTN [15] es una modificación de GPU BLAST para aminoácidos y también enfocado en la segunda fase Smith-Waterman, HCUDABLAST [16] que a diferencia de GPU BLAST usa un entorno con *Hadoop*, cuBLASTP [17] paraleliza NCBI-BLASTP enfocado en la fase de búsqueda de secuencias o Smith-Waterman usando GPU, H-BLAST [18] paraleliza NCBI-BLASTP y NCBI-BLASTX usando GPU enfocandose en la segunda fase Smith-Waterman.

Finalmente existen versiones que modifican la forma como se obtiene el "***seed index***" como: HS-BLASTN [19] que acelera la búsqueda en NCBI-BLASTN al generar un nuevo índice de semillas "*seed index*" a través de un *FM-INDEX* generado por el algoritmo *Burrows-Wheeler transform* BWT derivada de la base de datos, MerAligner [20] que es una nueva herramienta basada en BLAST que genera una tabla hash distribuida como *seed index* basada su estructura en una matriz esparsa y una paralelización en una supercomputadora Cray XC30.

Todas estas aplicaciones optimizan el tiempo de ejecución del BLAST y hacen uso de diversas bibliotecas y plataformas, sin embargo no han optimizado el algoritmo BLAST en la construcción del seed index usando GPU.

III. Propuesta

Optimizar en tiempo de ejecución la primera fase o "**seed index**" del algoritmo BLAST usando GPU con CUDA, construyendo una Tabla Hash basada en una matriz bidimensional donde la clave hash será una representación numérica en base a la *K-mer sequence*, obteniendose el acceso a los punteros de forma directa sin realizar búsquedas secuenciales o aleatorias de la *K-mer* o usando un *FM-INDEX* cuyo pseudo-código se muestra en el "Algoritmo 1".

Algorithm 1 pseudo-code: Optimization of BLAST seed indexing

Input: Pre-processed reference $sequence$, $SIZE\_KEY$, $BLOCK\_SIZE$
Output: two-dimensional Hash Table filled with seed index

```
1: length_seq ← Length reference sequence
2: Copy sequence from host memory to device memory
3: Create Matrix with Max_Length_Hash
4: while index is in range do
5:    share block memory BLOCK_SIZE + SIZE_KEY -1
6:    synchronize GPU
7:    HashKey[index] ← 0
8:    fact ← 1
9:    for i = SIZE_KEY − 1 to 0 do
10:      HashKey[index] ← HashKey[index] + fact ∗ sequence[index + i]
11:      fact ← fact ∗ 5
12:   end for
13: end while
14: Copy HashKeys from device memory to host memory
15: for i = 0 to length_seq − SIZE_KEY + 1 do
16:   Matrix[HashKey[i]] ← Insert i
17: end for
```

IV. GPU y CUDA

CUDA (*Compute Unified Device Architecture*) es una arquitectura de cálculo paralelo de NVIDIA, usado en la GPU (unidad de procesamiento gráfico) para incrementar el rendimiento del sistema.

Los programas CUDA contienen una parte secuencial, llamada kernel. La unidad básica de operación es el hilo ("thread"), los cuales estan organizados en bloques ("blocks") y estos a su vez estan organizados en grillas ("grids"), donde una grilla puede ejecutar un kernel.

Los hilos de un bloque se pueden identificar utilizando 1 Dimensión (x), 2 Dimensioness (x, y) o 3 Dimensiones (x, y, z). Así cada hilo tiene una identificación única asociada (threadIdx, blockIdx) ϵ {0, ..., dimBlock-1} x {0, ..., dimGrid-1}. Indicando el ID dentro de su bloque de hilos (threadIdx)

y la ID del bloque de hilos dentro de la grilla (blockIdx). De forma similar a los procesos MPI, CUDA proporciona acceso a cada subproceso a su identificación única a través de las variables correspondientes. El tamaño total de una cuadrícula (dimGrid) y un bloque de subprocesos (dimBlock) se especifica explícitamente en la función de llamada al kernel: $<<< dimGrid, dimBlock, configs >>> (Parametros)$

V. Metodología

A. Base de Datos

Las secuencias de ADN utilizadas fueron obtenidas de los portales de acceso público: NCBI [21] y EMSEMBL [22], que brinda archivos en formato FASTA, cuya estructura se observa en la Fig. 1, para lo cual se ha extraído exclusivamente la secuencia de bases sin considerar los comentarios, generando el indice de semillas ("*seed index*") de secuencias de archivos FASTA desde 50 kb hasta 1.3 Gb que es el genoma completo de la gallina (*Gallus gallus*) que posee 1 230 258 557 pares de bases pb.

```
>gi|263191547|ref|NM_000249.3| Homo sapiens mutL homolog 1 (MLH1), transcript variant 1, mRNA
GAAGAGACCCAGCAACCCACAGAGTTGAGAAATTTGACTGGCATTCAAGCTGTCCAATCAATAGCTGCCGCTGAAGGGTG
GGGCTGGATGGCGTAAGCTACAGCTGAAGGAAGAACGTGAGCACGAGGCACTGAGGTGATTGGCTGAAGGCACTTCCGTT
GAGCATCTAGACGTTTCCTTGGCTCTTCTGGCGCCAAAATGTCGTTCGTGGCAGGGGTTATTCGGCGGCTGGACGAGACA
GTGGTGAACCGCATCGCGGCGGGGGAAGTTATCCAGCGGCCAGCTAATGCTATCAAAGAGATGATTGAGAACTGTTTAGA
TGCAAAATCCACAAGTATTCAAGTGATTGTTAAAGAGGGAGGCCTGAAGTTGATTCAGATCCAAGACAATGGCACCGGGA
TCAGGAAAGAAGATCTGGATATTGTATGTGAAAGGTTCACTACTAGTAAACTGCAGTCCTTTGAGGATTTAGCCAGTATT
TCTACCTATGGCTTTCGAGGTGAGGCTTTGGCCAGCATAAGCCATGTGGCTCATGTTACTATTACAACGAAAACAGCTGA
TGGAAAGTGTGCATACAGAGCAAGTTACTCAGATGGAAAACTGAAAGCCCCTCCTAAACCATGTGCTGGCAATCAAGGGA
CCCAGATCACGGTGGAGGACCTTTTTTACAACATAGCCACGAGGAGAAAAGCTTTAAAAAATCCAAGTGAAGAATATGGG
AAAATTTTGGAAGTTGTTGGCAGGTATTCAGTACACAATGCAGGCATTAGTTTCTCAGTTAAAAAACAAGGAGAGACAGT
AGCTGATGTTAGGACACTACCCAATGCCTCAACCGTGGACAATATTCGCTCCATCTTTGGAAATGCTGTTAGTCGAGAAC
TGATAGAAATTGGATGTGAGGATAAAACCCTAGCCTTCAAAATGAATGGTTACATATCCAATGCAAACTACTCAGTGAAG
AAGTGCATCTTCTTACTCTTCATCAACCATCGTCTGGTAGAATCAACTTCCTTGAGAAAAGCCATAGAAACAGTGTATGC
```

Fig. 1. Secuencia de ADN en formato FASTA de *Homo sapiens* mutL homolog 1 (MLH1)

B. Pre-procesamiento

Al procesar secuencias de genomas, además de las bases conocidas **A** (adenina), **C** (citocina), **G** (guanina) y **T** (timina) es usual encontrar el caracter **N** que representa una base hipotética que no ha sido identificada en dicha posición, por lo que para su adecuado procesamiento de datos es necesario realizar una codificación como preprocesamiento convirtiendo las bases a su representación numérica siendo: N=0, A=1, C=2, G=3 y T=4.

C. Fase de Sembrado e indexamiento

El algoritmo BLAST [4] es un método heurístico, donde se va a procesar toda la secuencia de referencia ("db"), considerando la longitud de la clave o *k-mer* del índice de semillas como el número de las primeras pb tomadas para su indexamiento, las cuales son agrupadas por similitud como se observa en la "figura 2".

Dentro de esta fase podemos diferenciar dos partes, las cuales son: la generación de las claves Hash y la inserción de los punteros dentro de la Tabla Hash según su clave Hash. La optimización presentada se realiza exclusivamente en la generación de la clave Hash donde la similitud de secuencias se da por una igualdad en sus claves Hash, quedando la

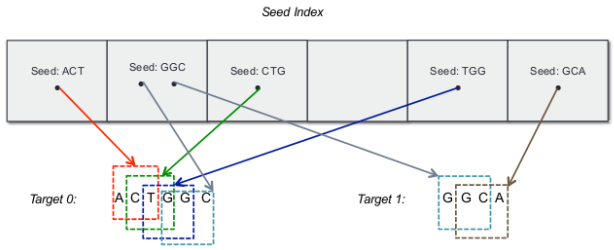

Fig. 2. Fase de sembrado e indexamiento del algoritmo BLAST [20]

inserción en la Tabla Hash como un algoritmo secuencial, el resultado de esta fase optimizada se puede observar en la "figura 3".

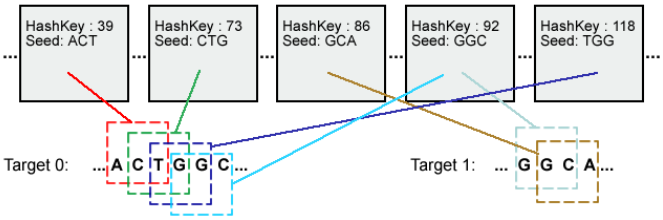

Fig. 3. Fase de sembrado e indexamiento del algoritmo BLAST Optimizado

La clave Hash generada está definida por la "ecuación 1" donde K es el k-mer de posición i, n es la longitud de la semilla *seed* y B es el número de bases consideradas (A, C, G, T y N) que en este caso es $B = 5$.

$$Hash\_Key = \sum_{i=0}^{n-1} K_i B^{(n-1)-i} = \sum_{i=0}^{n-1} K_i 5^{(n-1)-i} \quad (1)$$

La longitud de la Tabla Hash generada está definida por la "ecuación 2" donde se halla el máximo valor que puede poseer una clave Hash tomando como referencia la "ecuación 1", reemplazando los k-mer por su máximo valor: $K = 4$, dependiendo exclusivamente de la longitud de la clave n.

$$Length\_Hash\_Table = \left(\sum_{i=0}^{n-1} 4 * 5^{(n-1)-i} \right) + 1 \quad (2)$$

D. Fase de Extensión con Smith-Waterman

Para esta fase se procesan las secuencias de consulta "*query*", preprocesando su secuencia seed y expresandola en números, para luego hallar su clave Hash de acuerdo a la "ecuación 1". Posteriormente se aplica el algoritmo de alineamiento Smith-Waterman [7] entre la secuencia de consulta "*query*" y todas las semillas o seed que poseen la misma clave Hash dentro de la Tabla Hash, con la finalidad de identificar las que posean mayor score o hits de coincidencia.

Existen diversas optimizaciones del algoritmo de Smith Waterman como el CUDASW++ [23] usando CUDA y SIMD. En este trabajo no desarrollamos una optimización de este algoritmo por lo que se consideró una implementación simple del SW.

E. Evaluación de la optimización

La evaluación de la optimización se realizó usando pruebas en una laptop con procesador Intel $Core^{TM}$ i7-7500U CPU @ 2.70GHz x 4 , con 16 GB de memoria RAM y una tarjeta gráfica GForce 940MX de 384 núcleos con 2GB de memoria dedicada, donde se procesaron secuencias de ADN en formato FASTA de diferentes longitudes de pb en tres implementaciones: una secuencial, otra usando OPENMP y la tercera usando GPU.

VI. RESULTADOS Y DISCUSIÓN

En la implementación con GPU, las hebras son organizadas en bloques que para las pruebas fueron organizadas en bloques de 16 hebras cada uno, donde previo al procesamiento se extrae de la memoria global de GPU a la memoria por bloque sólo la secuencia que utilizarán las hebras dentro del bloque para generar la clave hash. Posterior a ello se genera la clave Hash de acuerdo a la "ecuación 1", almacenando el resultado en el array correspondiente.

La llamada al kernel o inicialización del código se dió con el "algoritmo 2".

Algorithm 2 code: Kernel initialization

$seed <<< (lengthDB - SIZE\_KEY)/BLOCK\_SIZE, BLOCK\_SIZE >>> (d\_A, d\_H);$

En la generación de las claves Hash, el algoritmo implementado con GPU obtuvo un tiempo de ejecución menor en comparación al implementado de forma secuencial, reduciendo un 18.7% su tiempo de ejecución durante la generación de claves hash al procesar el genoma completo de *Gallus gallus* con una longitud de la clave o "SIZE_KEY" de 5 pb, como se observa en la "figura 4". Asímismo se observa que la implementación con OPENMP presenta tiempos de ejecución elevados en referencia a las otras dos implementaciones lo cual evidencia el costo de acceso a la memoria por las diferentes hebras desplegadas en esta implementación.

Con una longitud de la clave hash o "SIZE_KEY" de 5 pb se logra una optimización de hasta un 11.24% del tiempo de ejecución en comparación con la implementación secuencial considerando el tiempo total del indexamiento de semillas (tiempo de generación de semillas + tiempo de inserción de los punteros dentro de la Tabla Hash) del algoritmo BLAST como se observa en la "Figura. 5".

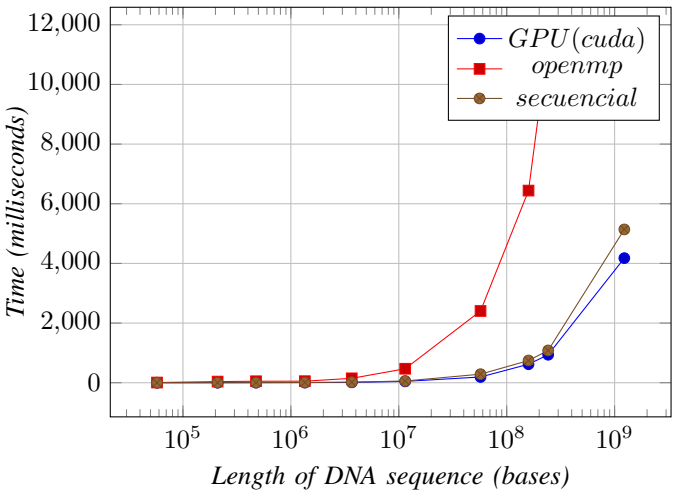

Fig. 4. Processing time vs. length of DNA reference sequence getting the hash keys with size_key = 5pb

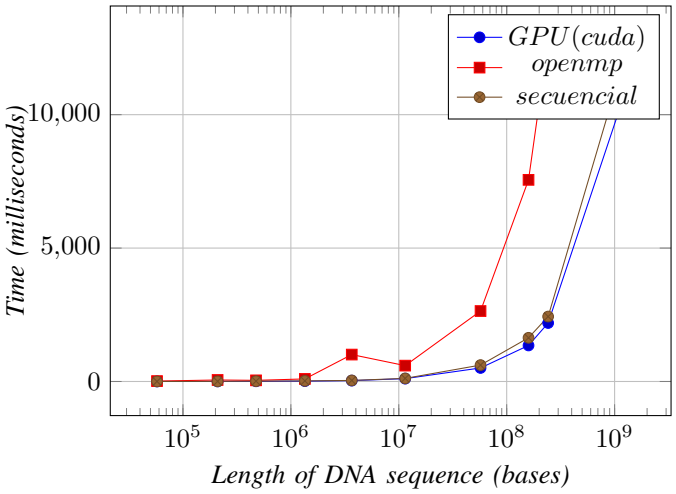

Fig. 5. Processing time vs. length of DNA reference sequence getting Hash Table of seed index with size_key = 5pb

Asimismo al realizar la comparación de tiempos de ejecución variando la longitud de la clave hash o "SIZE_KEY" procesando la secuencia completa del genoma de *Gallus gallus* se muestra un mejor rendimiento en el tiempo de ejecución de la implementación con GPU que la implementación secuencial si su longitud de la clave hash es mayor a 4 como se observa en la "Figura 6".

Al comparar, el indexamiento FM-INDEX utilizando el algoritmo BWT [19], para encontrar las ocurrencias (occ) de un patrón P[1..p] en una secuencia S[1..u] lo hace con un complejidad en tiempo de $O(p + occ * log^{\epsilon} u)$. Sin embargo al utilizar este algoritmo la complejidad sería de O(1) pues la ubicación de todos los patrones de longitud p son encontrados, ordenados y almacenados por su función hash dentro de la tabla Hash, cuya posición en la tabla está relacionada

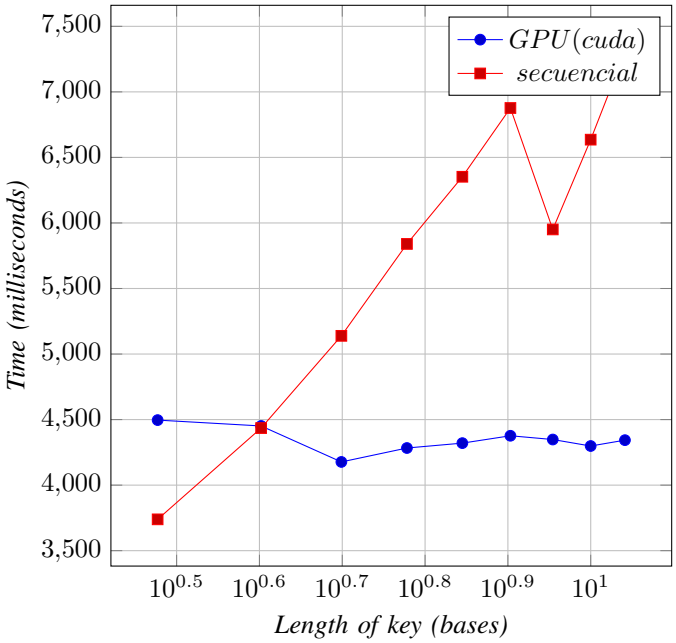

Fig. 6. Processing time vs. length size_key on DNA sequence of *Gallus gallus* getting the hash keys.

con el patrón, pudiendo generar un índice de semillas para cualquier tipo de genoma con la misma eficiencia que la prueba realizada.

.

VII. Conclusiones

La implementación con GPU usando CUDA propuesta para el indexamiento de semillas del algoritmo BLAST presenta un mejor rendimiento a medida que la longitud de la clave o "key" asciende, logrando una reducción de 11.24% del tiempo de ejecución al procesar el genoma completo de la gallina (*Gallus gallus*) con una longitud de la clave hash de 5 pb en comparación con la implementación secuencial, con una complejidad en tiempo de $O(1)$ para obtener semillas idénticas a la clave patrón de múltiples secuencias de consulta ("query").

Recomendaciones

Al procesar grandes cantidades de datos de secuencias de ADN reales, se debe tener en consideración la memoria RAM disponible y la memoria dedicada del GPU que se puede utilizar, para poder realizar ciclos de procesamiento como una forma de reducir el consumo de la RAM y la memoria usada de la GPU.

Esta optimización fue realizada y evaluada de forma local, siendo de gran interés su evaluación en entornos distribuidos y con tarjetas gráficas de mayor número de núcleos para evaluar su rendimiento en comparación con otras versiones optimizadas del BLAST existentes.

Agradecimientos

A la *Universidad Nacional de San Agustín*, UNSA, por el financiamiento del proyecto de investigación según contrato Nº *TT-0150-2016*.

References

[1] Mardis, E. R. (2008). Next-generation DNA sequencing methods. Annu. Rev. Genomics Hum. Genet., 9, 387-402.
[2] Li, H., & Homer, N. (2010). A survey of sequence alignment algorithms for next-generation sequencing. Briefings in bioinformatics, 11(5), 473-483.
[3] Ticona, W. G. C. (2003). Aplicao de Algoritmos Genticos Multi-Objetivo para Alinhamento de Seqłncias Biolgicas. Instituto de Cilncias Matemticas e Computao, Universidade de So Paulo, So Carlos, SP.
[4] Altschul, S. F., Gish, W., Miller, W., Myers, E. W., & Lipman, D. J. (1990). Basic local alignment search tool. Journal of molecular biology, 215(3), 403-410.
[5] Eddy, S. R. (2011). Accelerated profile HMM searches. PLoS computational biology, 7(10), e1002195.
[6] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., & Hwu, W. M. W. (2008, February). Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (pp. 73-82). ACM.
[7] Smith, T. F., & Waterman, M. S. (1981). Comparison of biosequences. Advances in applied mathematics, 2(4), 482-489.
[8] Krishnan, A. (2005). GridBLAST: a Globusbased highthroughput implementation of BLAST in a Grid computing framework. Concurrency and Computation: Practice and Experience, 17(13), 1607-1623.
[9] Cushman, J. C., & Yim, W. C. (2017). Divide and Conquer (DC) BLAST: fast and easy BLAST execution within HPC environments.
[10] Matsunaga, A., Tsugawa, M., & Fortes, J. (2008, December). Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications. In eScience, 2008. eScience'08. IEEE Fourth International Conference on (pp. 222-229). IEEE.
[11] Lin, H., Ma, X., Feng, W., & Samatova, N. F. (2011). Coordinating computation and I/O in massively parallel sequence search. IEEE Transactions on Parallel and Distributed Systems, 22(4), 529-543.
[12] Darling, A. E., Carey, L., & Feng, W. C. (2003). The design, implementation, and evaluation of mpiBLAST (No. LA-UR-03-2862). Los Alamos National Laboratory.
[13] Sawyer, S. E., Rekepalli, B., Horton, M. D., & Brook, R. G. (2015, September). HPC-BLAST: distributed BLAST for Xeon Phi clusters. In Proceedings of the 6th ACM Conference on Bioinformatics, Computational Biology and Health Informatics (pp. 512-513). ACM.
[14] Vouzis, P. D., & Sahinidis, N. V. (2010). GPU-BLAST: using graphics processors to accelerate protein sequence alignment. Bioinformatics, 27(2), 182-188.
[15] Zhao, K., & Chu, X. (2014). G-BLASTN: accelerating nucleotide alignment by graphics processors. Bioinformatics, 30(10), 1384-1391.
[16] Khare, N., Khare, A., & Khan, F. (2017). HCudaBLAST: an implementation of BLAST on Hadoop and Cuda. Journal of Big Data, 4(1), 41.
[17] Zhang, J., Wang, H., & Feng, W. C. (2017). cublastp: Fine-grained parallelization of protein sequence search on cpu+ gpu. IEEE/ACM transactions on computational biology and bioinformatics, 14(4), 830-843.
[18] Ye, W., Chen, Y., Zhang, Y., & Xu, Y. (2017). H-BLAST: a fast protein sequence alignment toolkit on heterogeneous computers with GPUs. Bioinformatics, 33(8), 1130-1138.
[19] Chen, Y., Ye, W., Zhang, Y., & Xu, Y. (2015). High speed BLASTN: an accelerated MegaBLAST search tool. Nucleic acids research, 43(16), 7762-7768.
[20] Georganas, E., Bulu, A., Chapman, J., Oliker, L., Rokhsar, D., & Yelick, K. (2015, May). meraligner: A fully parallel sequence aligner. In Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International (pp. 561-570). IEEE.
[21] NCBI. [En línea]. Disponible en: https://goo.gl/rRirdm. [Accedido: 05-abr-2018]

[22] EMSEMBL. [En línea]. Disponible en: https://goo.gl/HJDEqq. [Accedido: 10-feb-2018]
[23] Liu, Y., Wirawan, A., & Schmidt, B. (2013). CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. BMC bioinformatics, 14(1), 117.