UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Grado en Ingeniería Informática en Ingeniería de Computadores

Trabajo Fin de Grado

Resolución del Problema de Búsqueda de Epistasia mediante Paralelismo y Computación Heterogénea

Autor: Jorge Montero De Amuedo

Fecha:

UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Grado en Ingeniería Informática en Ingeniería de Computadores

Trabajo Fin de Grado

Resolución del Problema de Búsqueda de Epistasia mediante Paralelismo y Computación Heterogénea

Autor: Jorge Montero De Amuedo

Tutores: José María Granado Criado y Sergio Santander Jiménez

Tribunal Calificador

Presidente:

Secretario:

Vocal:

ÍNDICE DE CONTENIDOS

1 INTRODUCCIÓN 3

1.1 Epistasia **3**

1.2 Problemática de la Epistasia **4**

1.3 Organización del Documento **6**

2 OBJETIVOS 7

2.1 Objetivos Fijados **7**

2.2 Planificación de Tareas a Realizar **8**

3 ANTECEDENTES / ESTADO DEL ARTE 9

3.1 Métodos de Búsqueda Exhaustiva **9**

3.1.1. Modelos de Regresión Paramétrica 9

3.1.1.1 Regresión Logística 10

3.1.1.2 Técnicas de Regresión Penalizadas 10
3.1.2. Representación Bit a Bit de Datos y Pruebas basadas en la Ratio de Probabilidad 11
3.1.3. Análisis de Curva ROC (Característica Operativa de Receptor) 12
3.1.4. Reducción de Dimensionalidad Multifactorial (MDR) 13

3.2 Métodos de Búsqueda Heurística 14

3.3 Métodos de Búsqueda Utilizando Machine Learning 15

4 MATERIAL Y MÉTODO 18

4.1 Traducción de Matlab a C++ 19

4.2 Implementación de OpenCL 22
4.2.2. Organización de Memoria para GPU y CPU 25

4.3 Implementación de OpenMP 26

4.4 Selección de Pruebas a Realizar y Descripción de los Datos Escogidos 27

5 RESULTADOS Y DISCUSIÓN 29

5.1 Optimización de Parámetros para los Dispositivos CPU y GPU 30

5.2 Experimentación 32

6 CONCLUSIONES 36

6.1 Conclusiones del Estudio 36

6.2 Conclusiones Personales 37

GLOSARIO 39

BIBLIOGRAFÍA 43

Algoritmos Importantes de la Versión Final 46

ÍNDICE DE TABLAS

Tabla 1.1 – Ejemplo de número de combinaciones con una base de datos de 100 SNPs para diversos tamaños de k. 5

ÍNDICE DE FIGURAS

Figura 4.1 Acceso de la GPU a los datos en memoria. 25
Figura 5.1 – Resultados de la ejecución en el dispositivo GPU Nvidia Titan X en una base de datos de 1000 SNPs y 4000 pacientes. 30
Figura 5.2 - Resultados de la ejecución en el dispositivo CPU 2xIntel Xeon E5-2630 v3 en una base de datos de 1000 SNPs y 4000 pacientes. 31
Figura 5.3 – Tiempos de ejecución de las versiones secuencial y paralelas con una base de datos de 31341 SNPs y 146 pacientes. 33
Figura 5.4 – Tiempos de ejecución de las versiones paralelas con una base de datos de 31341 SNPs y 146 pacientes. 34

RESUMEN

En este estudio se hablará de la problemática del análisis de la epistasia y del problema de carga computacional que lleva consigo el estudio de grandes conjuntos de datos. También se hablará de algunas de las soluciones software basadas en diferentes tipos de algoritmos que se han utilizado para resolver este problema. Por otro lado se presentará una solución propia de búsqueda exhaustiva, tomando como base la implementación ESMO realizada en MATLAB, que podemos encontrar en (Li, 2017). A partir de dicha solución se realizará una implementación paralela y heterogénea, es decir, que se pueda ejecutar en varios dispositivos diferentes (CPU y GPU) de forma simultánea, aprovechando toda la potencia computacional de éstos. Además, se realizará un estudio sobre la ganancia en tiempo al aprovechar la computación heterogénea.

ABSTRACT

In this study we will talk about the problem of the analysis of epistasis and the problem of computational load that involves the study of large data sets. We will also talk about some of the software solutions based on different types of algorithms that have been used to solve this problem. On the other hand, we will present our own exhaustive search solution, based on the ESMO solution implemented in MATLAB, which can be found in (Li, 2017). From this solution we will perform a heterogeneous parallel implementation, that is, it can be run on several different devices (CPU and GPU) simultaneously, taking advantage of all the computing power of these devices. In addition, an execution time study will be conducted to analyze the advantage of heterogeneous computing.

1

INTRODUCCIÓN

1.1 Epistasia

Se conoce como Epistasia al fenómeno que se produce por la interacción entre dos o más genes cuando un gen enmascara el fenotipo (o efecto) de otro gen en un locus [14] (posición fija de un cromosoma que determina la posición de un gen) diferente del primero. Al gen cuyo fenotipo enmascara al del otro se le llama epistásico, por otro lado, al gen cuyo fenotipo es suprimido o alterado se le llama hipostásico.

Actualmente se conocen los siguientes casos de epistasias según los genes epistásicos sean dominantes o recesivos: Epistasia Simple Recesiva, donde uno de los locus, que tiene un gen con dos alelos recesivos, suprime la expresión genética de otro; Epistasia Simple Dominante, donde uno de los locus, que tiene un gen con un alelo dominante, suprime la expresión genética de otro; Epistasia Doble Recesiva, donde

cualquiera de los dos locus, con ambos alelos recesivos, pueden suprimir la expresión genética de otro.

En la actualidad, el estudio genético de la epistasia de los polimorfismos de un solo nucleótido o SNP (*Single Nucleotide Polymorphism*) [23], que se localizan en cualquier parte de la estructura de los genes y el genoma [13], y que representan las variantes genéticas más comunes encontradas en el ser humano, puede ayudarnos a descubrir por qué se producen y cómo tratar enfermedades complejas como el cáncer, la diabetes, el asma, o la artritis reumatoide, entre otras (Nickle & Barrette-Ng, 2019). Aunque estas enfermedades no tienen un patrón de herencia definido, como si lo tienen las enfermedades mendelianas (llamadas así por transmitirse a la descendencia según las leyes de Mendel), se sabe que, en estas patologías complejas, el componente genético y las variantes comunes de tipo SNP desempeñan un papel determinante en el desarrollo de éstas.

1.2 Problemática de la Epistasia

Al contrario que las enfermedades mendelianas, en las patologías complejas, debido a la ausencia de patrones de herencia, y a la falta de reproducibilidad [21], el estudio de las relaciones entre los SNPs (o locus) y las enfermedades complejas ha pasado de tratarse como una variación a tratarse como interacciones de biomarcadores [7] definidos como epistasias.

El análisis de las epistasias en todo el genoma tiene por delante tres retos fundamentales:

En primer lugar, tenemos el problema de que, si aumentamos el número de variables, es decir, de SNPs relacionados entre sí, el número de combinaciones aumenta de forma exponencial siguiendo la ecuación: SNPs! / (k! * (SNPs - k)!), donde k es igual al número de SNPs relacionados entre sí. La tabla 1.1 muestra los valores de las posibles combinaciones para tamaños de k entre 2 y 11 en una base de datos con solo 100 SNPs.

K	Combinaciones
2	4950
3	161700
4	3921225
5	75287520
6	1192052400
7	16007560800
8	1,86088E+11
9	1,90223E+12
10	1,73103E+13
11	1,4163E+14

Tabla 1.1 – Ejemplo de número de combinaciones con una base de datos de 100 SNPs para diversos tamaños de k.

En segundo lugar, los numerosos biomarcadores de la epistasia se analizarán para determinar si la interacción entre ciertos SNPs está relaciona con enfermedades complejas desde el punto de vista estadístico, por lo que los resultados de la asociación pueden ser falsos positivos [12] y son difíciles de ser replicados.

En tercer lugar, desde el punto de vista del aprendizaje estadístico, la gran cantidad de SNPs y la escasez de muestras plantea un problema que provoca la falta de capacidad de generalización [9].

Para resolver estos problemas, en este proyecto se va a utilizar el método de búsqueda exhaustiva, es decir, se prueban todas las combinaciones posibles, entre todas las combinaciones de los SNPs de la base de datos con k igual a tres.

Para realizar la búsqueda exhaustiva se utilizará el modelo de programación paralela mediante la librería OpenCL [18] y la potencia de cómputo de CPUs con varios núcleos y GPUs.

1.3 Organización del Documento

Este documento se compone de varias entradas:

En primer lugar tenemos la introducción, donde hablaremos sobre la epistasia y por qué es importante su estudio, la problemática que conlleva, así como el enfoque general, las herramientas que vamos a utilizar para resolver este problema y la organización del este documento.

En segundo lugar hablaremos de los objetivos de este Trabajo de Fin de Grado, en el que buscamos resolver el problema de la epistasia de 3 loci (k = 3) utilizando la paralelización con OpenCL en GPUs y CPUs, y también hablaremos de la planificación de tareas a realizar derivadas de éstos.

En tercer lugar hablaremos de los antecedentes y el estado del arte, donde explicaremos cómo se está resolviendo este problema en la actualidad, y explicaremos algunos artículos como ejemplos de distintas soluciones.

En cuarto lugar hablaremos del material y el método, es decir, contaremos todo lo relacionado con nuestra solución, explicando el trabajo realizado en ésta.

En quinto lugar hablaremos de los resultados obtenidos por nuestra solución.

Por último, tendremos un apartado de conclusiones.

2

OBJETIVOS

2.1 Objetivos Fijados

El objetivo de este Trabajo de Fin de Grado es resolver la problemática de la epistasia de 3 loci, es decir, el problema de la epistasia con 3 SNPs relacionados entre sí, expresado de forma matemática como: SNPs! / (k! * (SNPs - k)!) con k = 3, donde dicha solución se probará con bases de datos de SNPs de distintos tamaños.

El objetivo principal es desarrollar una solución lo más rápida, eficiente, flexible, y robusta posible, utilizando el paradigma del paralelismo heterogéneo con la potencia computacional de las GPUs y la CPUs multinúcleo.

Para esto deberemos desarrollar una solución al problema de la epistasia de 3 loci que nos permita parametrizar la base de datos de SNPs de distintos tamaños, y también que podamos lanzarla en distintos dispositivos para que nos permita una computación

heterogénea, es decir, que utilice procesadores de tipo CPU y GPU para aprovechar toda la potencia de cómputo posible.

También debemos paralelizar el número de hilos de cómputo que queremos lanzar en la ejecución del programa, y el número de elementos de cómputo que queremos que ejecute cada hilo.

Para poder realizar esto, tendremos que elegir las herramientas y librerías de software más adecuadas.

Por último, tendremos que realizar un estudio para describir qué parámetros anteriormente comentados son los más adecuados para resolver el problema de la epistasia de 3 loci con la mayor eficiencia en tiempo, y también la eficiencia que conseguimos ejecutando el programa en más de un dispositivo.

2.2 Planificación de Tareas a Realizar

Para poder cumplir estos objetivos, se han definido las siguientes tareas:

En primer lugar, se ha decidido partir de una solución a este problema que podemos encontrar en (Li, 2017), llamada ESMO. Esta solución está implementada en el lenguaje Matlab, y es necesario traducir el programa al lenguaje C++, e implementar una versión secuencial del mismo.

En segundo lugar, es necesario implementar una versión que habilite la computación paralela con un dispositivo (CPU o GPU) partiendo de la versión secuencial anterior, para esto se ha decidido utilizar OpenCL en la versión paralela del programa.

En tercer lugar, y partiendo de la versión paralela del programa, es necesario implementar una versión que permita aprovechar la potencia computacional de varios dispositivos (CPU o/y GPU) de forma simultánea, para realizar esta tarea, se ha decidido utilizar OpenMP [19] para realizar la gestión de los dispositivos también de forma paralela

Una vez tengamos todas las versiones del programa, podemos realizar el estudio.

3

ANTECEDENTES / ESTADO DEL ARTE

3.1 Métodos de Búsqueda Exhaustiva

Este método se basa en probar exhaustivamente todas las combinaciones posibles, y es la solución que hemos adoptado nosotros en este proyecto. A continuación, vamos a ver diferentes alternativas existentes para esta metodología.

3.1.1. Modelos de Regresión Paramétrica

El modelo de regresión paramétrica es un modelo estadístico que se basa en suposiciones que estén lo más cerca posible de la realidad sobre la distribución de probabilidad que genera los datos. Por lo tanto, cuanto más realistas sean estas

suposiciones, más precisas serán las predicciones. Sin embargo, si las suposiciones utilizadas son incorrectas el modelo no funcionará. Los algoritmos utilizados en el modelo de regresión paramétrica tienen un número fijo de parámetros que se estiman a partir de los datos. A continuación, vemos algunos modelos utilizados en este tipo de solución.

3.1.1.1 Regresión Logística

En primer lugar, tenemos el modelo de regresión logística que ha sido utilizado para la búsqueda exhaustiva de interacciones entre todas las combinaciones.

Éste es un modelo estadístico que representa la relación entre una combinación lineal de variables, en este caso un SNP y un rasgo binario (si un SNP afecta al fenotipo o no, donde el fenotipo representa la enfermedad).

Un ejemplo de este modelo es el Software PLINK (Purcell, et al., 2007), que ha implementado modelos de regresión logística para el análisis de epistasia. Sin embargo, este modelo no es recomendable para estudios con un gran conjunto de datos, ya que la estimación de los parámetros necesarios es un procedimiento muy costoso y poco preciso, introduciendo muchos errores debido a que el tamaño de muestra es muy pequeño en comparación con el tamaño de datos de todo el genoma. Por este motivo, se genera un gran número de falsos positivos.

Este problema se intenta resolver corrigiendo la probabilidad de que un SNP afecte a una enfermedad utilizando el método de corrección de prueba múltiple de Bonferroni[15]. Sin embargo, esta corrección es demasiado conservadora y solo se detectan interacciones con efectos muy fuertes, por lo que se perderán muchas otras interacciones.

3.1.1.2 Técnicas de Regresión Penalizadas

Estas técnicas de regresión paramétrica tratan de analizar interacciones entre pares de SNPs de forma bidireccional (k = 2).

Algunos ejemplos de estas técnicas son "*Least Absolute Shrinkage and Selection Operator*" (LASSO) (Kukreja, et al., 2016), que realiza tanto la selección de variables como la regularización para mejorar la precisión de predicción y la interpretabilidad

del modelo estadístico, o "*Smoothly Clipped Absolute Desviation*" (SCAD) (Kim, et al., 2007), que estima los parámetros mientras selecciona simultáneamente variables importantes y que, en comparación con el anterior, no solo selecciona variables importantes de manera consistente, sino que también produce estimadores de parámetros tan eficientes como si se conociera el modelo verdadero. A esta característica se denomina "estimador de oráculo".

Desafortunadamente, estas técnicas son propensas a la tasa de falsos positivos inflados y tienen un coste computacional demasiado elevado para buscar exhaustivamente en todo el espacio de búsqueda de interacción por pares.

Un ejemplo del uso de estas técnicas para el análisis de epistasia lo encontramos en (Gou, et al., 2014).

3.1.2. Representación Bit a Bit de Datos y Pruebas basadas en la Ratio de Probabilidad

Un ejemplo de este modelo podemos encontrarlo en (Wan, et al., 2010). Esta técnica se basa en el software BOOST, donde primero se calcula, para cada par de SNPs, una tabla de contingencia [25] de tamaño 3x3 con la distribución de frecuencias de los nueve genotipos posibles. Sin embargo, calcular todas las tablas de contingencia de todos los pares de SNPs del genoma es un proceso muy lento. Para paliar este problema, los datos del genoma se transforman primero a una forma binaria, de forma que, al contrario que en la representación de datos habitual donde cada fila simboliza un SNP y cada columna simboliza un sujeto (o viceversa), en la representación binaria cada SNP está representado por una tabla de tres filas por dos columnas, donde la posición de la fila representa el estado del genotipo (0, 1, o 2), una columna que representa los casos, y otra que representa el control de sujetos. Cada celda de la tabla contiene una cadena de bits donde cada bit representa un sujeto y su genotipo, si corresponde al genotipo codificado por la fila actual el bit tendrá valor 1, el bit tendrá valor 0 en caso contrario. De esta forma la matriz resultante es mucho menor ya que todos los valores de la tabla son binarios.

Esa representación también se mantiene cerca del lenguaje de máquina, lo que significa que construir estas tablas de contingencia a partir de ella solo implica operaciones rápidas a nivel de bits (es decir, booleanas).

Una vez tenemos las tablas de contingencia para cada par de SNPs se prueba una desviación del modelo aditivo lineal [11]. Esta desviación se expresa en términos de probabilidades logarítmicas bajo el supuesto de equivalencia entre un modelo de regresión logística y su modelo de log-verosimilitud (función que determina los valores óptimos de los coeficientes estimados) correspondiente. A diferencia del modelo tradicional de efecto marginal que se construye mediante iteraciones computacionalmente muy costosas, los autores utilizan una aproximación no iterativa de la relación de log-verosimilitud llamada aproximación de superposición de Kirkwood (KSA) [6]. Sobre la base de tablas de contingencia, todas las interacciones por pares se prueban con esta relación de log-verisimilitud. Sin embargo, utilizando esta relación de log-verosimilitud se siguen obteniendo falsos positivos, por lo que, tras esta primera fase de detección, a las combinaciones resultantes se le aplica otra fase de detección con una relación de probabilidad logarítmica clásica. Por último, BOOST utiliza la función estadística χ^2 (prueba estadística donde si la hipótesis es nula, es cierta) para evaluar en última instancia la importancia de las interacciones epistáticas, donde los pares SNP probados ya muestran una asociación significativa con una diferencia de probabilidad logarítmica entre el modelo que no considera las interacciones (modelo reducido) y el modelo que las considera (modelo completo).

3.1.3. Análisis de Curva ROC (Característica Operativa de Receptor)

Un ejemplo de este modelo lo podemos encontrar en (Goudey, et al., 2013), donde se presenta GWIS, que compara los SNPs por pares como en el modelo anterior, es decir k=2, pero a diferencia de éste, el modelo de curva ROC no se basa en el análisis de regresión, sino en analizar las curvas de características operativas del receptor (ROC) [8].

GWIS construye, para cada par de SNPs, tres modelos de clasificación de predicción basados en curvas ROC, uno para cada SNP del par de forma independiente, y uno para el par SNP. Cuando la curva ROC correspondiente a un par SNP se encuentra sobre las otras dos curvas que corresponden a los SNPs por separado, se dice que el par SNP tiene un mejor poder de predicción que los SNPs por separado. La problemática que se nos presenta es saber si esta desviación de poder de predicción entre las curvas ROC es significativa. Para resolver este problema, en el artículo se

propone una prueba de hipótesis sin modelo llamada diferencia en sensibilidad y especificidad (DDS), que cuantifica la ganancia de sensibilidad y especificidad [22] de una curva ROC sobre otra.

3.1.4. Reducción de Dimensionalidad Multifactorial (MDR)

Este modelo es uno de los más utilizados en la actualidad, ya sea el modelo completo, o como base para nuevos modelos.

En este modelo no es necesario estimar parámetros, lo cual es una ventaja respecto a los modelos anteriores, ya que no necesita de suposiciones sobre el modelo genético.

También aventaja a los modelos anteriores en que puede buscar tanto interacciones entre pares como interacciones de orden superior.

MDR divide el conjunto de datos en dos partes, la primera parte (conjunto de entrenamiento), es mucho mayor que la segunda y se utilizada para construir el modelo de interacción de los SNPs, y la segunda parte (conjunto de pruebas), se utiliza para evaluar el modelo construido.

Este modelo crea tablas de contingencia donde las celdas son todas las posibles combinaciones de interacciones de orden n de los SNPs. Después, se realiza el recuento de casos y controles para cada combinación de genotipo y cada celda se evalúa según la relación entre número de casos que comparten esta combinación de genotipo y número de controles que comparten esta combinación de genotipo, clasificando así la celda como de alto riesgo si supera un umbral especificado o de bajo riesgo en caso contrario. Tras este paso, el modelo fusiona las celdas marcadas como de alto riesgo en un grupo y las celdas marcadas de bajo riesgo en otro, de ahí el nombre “Reducción de dimensionalidad”, consiguiendo así un problema en el que la dimensión es igual al orden de interacción elegido.

Cada combinación de SNPs da como resultado un modelo de predicción. A continuación, un proceso de validación cruzada diez veces permite evaluar la calidad de dichos modelos. La proporción de individuos como afectados o no afectados se evalúa con la segunda parte del conjunto de datos (conjunto de pruebas). Durante las diez iteraciones se estima el error de predicción y al finalizar se conservan los mejores modelos. Un ejemplo de este método lo encontramos en (Moore & Andrews, 2014).

La característica principal de este método es la reducción de la dimensión, lo que hace que se pueda combinar con otros métodos de clasificación para generar un nuevo método de búsqueda de interacción de SNPs. Un ejemplo de esto lo encontramos en (Li, 2017). Este método se basa en la combinación del MDR con la búsqueda exhaustiva basada en la optimización de objetivos múltiples denominado "*Exhaustive Searching based on Multi-objective Optimization*" (ESMO). ESMO combina técnicas como la información mutua[26] o enfoques a redes bayesianas para evaluar combinaciones de SNPs epistáticas. Nuestro proyecto es una variante de ESMO, donde intentamos aprovechar toda la potencia de la computación heterogénea para obtener los mejores resultados de rendimiento y tiempo posibles. ESMO se explicará más en detalle en el apartado 4 "Material y Método".

3.2 Métodos de Búsqueda Heurística

Estos métodos utilizan algoritmos de búsqueda heurística, que tienen como objetivo reducir el espacio de búsqueda inspeccionando aquellas combinaciones que tengan más probabilidades de tener un resultado positivo.

Los algoritmos más utilizados en los métodos de búsqueda heurística son aquellos basados en redes bayesianas y combinaciones de éstos con otros algoritmos. Seguidamente veremos algunos ejemplos:

El primer ejemplo de este método de búsqueda que veremos lo encontramos en (Tuo, et al., 2016), donde encontramos el método FHSA-SED, que utiliza los algoritmos de búsqueda exhaustiva de redes bayesianas y el coeficiente de Gini [10] para clasificar dos loci SNP como modelo candidato. Para encontrar rápidamente los modelos candidatos más probables entre todos los modelos de dos loci se utiliza un algoritmo mejorado basado en el algoritmo de búsqueda armónica [3]. En cada modelo de dos loci se presenta una tabla bidimensional tabú [24] para evitar la evaluación repetida de algunos modelos de enfermedades que tienen un efecto marginal fuerte (pares de SNPs evaluados previamente y que se repiten a lo largo del algoritmo). Por último, para probar los modelos candidatos seleccionados se utiliza el modelo estadístico G-test [17].

El segundo ejemplo de método basado en la búsqueda heurística es el que podemos encontrar en (Han & Chen, 2011), donde nos presentan bNEAT, un algoritmo basado en redes bayesianas para la detección de interacciones de epistasia de SNPs. Este algoritmo es válido sobre todo para detectar interacciones epistáticas con efectos marginales leves o nulos.

El tercer ejemplo que veremos es el descrito en (Jing & Shen, 2014), donde se presenta MACOED como una metodología de optimización heurística multiobjetivo para detectar interacciones genéticas que combina los algoritmos de regresión logística y de redes bayesianas para clasificar los loci de los SNPs y detectar las interacciones. Esta combinación de algoritmos produce una tasa inferior de falsos positivos que si se detectaran las interacciones con estos mismos algoritmos de forma independiente. Para resolver el problema computacional en problemas de alta dimensión se utiliza una versión multiobjetivo del algoritmo de optimización de colonia de hormigas o "*Ant Colony Optimization*" basado en memoria integrado en MACOED, y que puede retener soluciones no dominadas encontradas en iteraciones pasadas.

El último ejemplo que veremos de método basado en redes bayesianas es el que podemos encontrar en (Aflakparast, et al., 2014). Este método, llamado búsqueda de epistasia de Cuco, detecta interacciones epistáticas significativas en estudios de asociación basados en la población con un diseño de casos y controles. Este método combina un algoritmo basado en redes bayesianas y otro algoritmo de búsqueda heurística basado en evolución para detectar interacciones. Este método se puede aplicar de manera eficiente en problemas de alta dimensión.

3.3 Métodos de Búsqueda Utilizando Machine Learning

La utilización de los métodos de aprendizaje automático es cada vez más frecuente en el análisis de epistasia con grandes conjuntos de datos. Sin embargo, la reproducibilidad de los análisis de aprendizaje automático del genoma puede verse obstaculizada por factores biológicos y estadísticos. Por este motivo, la mayoría de las metodologías que utilizan aprendizaje automático o *machine learning* tienen como objetivo optimizar las metodologías mencionadas anteriormente. A continuación veremos algunos ejemplos:

El primer ejemplo que veremos lo podemos encontrar en (Huang & Nogueira, 2018). Este método detecta la interacción de SNPs mediante un proceso de decisión de Markov [20], donde el estado es una representación latente codificada a partir de datos del genoma, y el espacio de acción son todos los SNPs. Las interacciones de los SNPs se pueden medir utilizando métodos como la "tasa de clasificación correcta" (CCR) de la "Reducción de dimensionalidad multifactorial" (MDR) y la "utilidad de regla" que podemos encontrar en (Yang, et al., 2018). Una vez tenemos el resultado, se seleccionan los SNPs que tienen una alta probabilidad de interacción a través de un umbral dado. Por último, un agente de aprendizaje de refuerzo (EpiRL *agent*) [1] aprenderá a seleccionar SNPs con una alta interacción utilizando el método de gradiente de políticas y optimizando así todo el proceso. Este aprendizaje se reforzará en cada iteración.

El segundo ejemplo es el que podemos encontrar en (Li, et al., 2018), donde nos proponen el método de aprendizaje profundo para análisis de epistasia y heterogeneidad o DEPH. Este método está dividido en tres etapas para el análisis, la detección y clasificación de epistasia. La primera etapa sería la detección de epistasia, que se realizaría aplicando un método de optimización multiobjetivo para encontrar varios conjuntos candidatos de SNPs epistáticos que contribuyen a diferentes subtipos de enfermedades complejas. En este caso se utiliza ESMO (apartado 3.1.4). En el segundo paso, se utiliza un algoritmo de agrupación K-means [2] para definir subtipos del grupo de casos. Por último, se utiliza un modelo de aprendizaje profundo para la predicción de enfermedades basado en la unidad de procesamiento de gráficos (GPU) utilizando un algoritmo de mini-lotes [4] escrito en el lenguaje de programación "Python". Este modelo es especialmente efectivo para el diagnóstico de enfermedades complejas cuando la epistasia y la heterogeneidad existen al mismo tiempo.

El tercer ejemplo que veremos es un método de validación cruzada [16] para la detección de epistasia. Este método es llamado "*Proportional Instance Cross Validation*" (PICV) o validación cruzada de instancias proporcionales y podemos encontrarlo en (Piette & Moore, 2018). A diferencia de otros métodos de validación cruzada, este método conserva la distribución original de una variable independiente al dividir el conjunto de datos en particiones de entrenamiento y prueba.

En el procedimiento tradicional de validación cruzada para cada escenario dado se realizan 1000 réplicas del procedimiento en el que dos tercios de las observaciones se

asignan aleatoriamente para su uso en el entrenamiento con aprendizaje automático y el tercio restante se usa para pruebas. Los datos obtenidos en el entrenamiento se utilizan para ajustar los modelos de regresión logística con y sin la interacción de pares de SNPs. Estos modelos se usan para predecir el estado de control de casos para los datos de pruebas almacenados, utilizando un umbral de 0,5 para la asignación de predicción de casos contra controles a partir de los valores ajustados. Estas predicciones se utilizan para calcular la sensibilidad, especificidad, valor predictivo positivo y valor predictivo negativo para los datos de prueba.

Por el contrario, en el procedimiento de validación cruzada de instancia proporcional, en lugar de asignar cada observación como entrenamiento o prueba de forma aleatoria, se asignan de manera específica al genotipo, de forma que dos tercios de las observaciones de cada genotipo SNP-SNP se asignan para su uso en el entrenamiento y el tercio restante se usa para pruebas, por lo que, aunque las proporciones de observaciones asignadas para entrenamiento y pruebas son las mismas que en el modelo tradicional, de esta forma las proporciones relativas de cada genotipo se conservan entre el conjunto de datos general y las particiones de entrenamiento y prueba. El resto del modelo es similar al tradicional.

PICV aborda la discordancia entre las particiones de entrenamiento y prueba que pueden ocurrir en la validación cruzada tradicional, por lo que mejora la consistencia entre las sensibilidades de entrenamiento y evaluación y los valores predictivos positivos. Por lo tanto, PICV es comparable a la validación cruzada tradicional en términos de especificidad y valor predictivo negativo al tiempo mejora en sensibilidad y valor predictivo positivo.

4

MATERIAL Y MÉTODO

Para realizar este proyecto, nos hemos basado en el proyecto ESMO que podemos encontrar en (Li, 2017). ESMO es una metodología basada en la optimización de multiobjetivo MDR que combina las metodologías basadas en entropía múltiple y redes bayesianas para evaluar combinaciones de SNPs epistáticas. Para detectar las combinaciones de SNPs con alta probabilidad de relación con enfermedades complejas, ESMO diseña un flujo de trabajo adaptativo basado en la ordenación de las combinaciones de SNPs no dominantes y la selección de estas a través de un algoritmo top-k [5] optimizado en tiempo. ESMO es una metodología no parametrizada y de tipo "model-free", o lo que es lo mismo, no requiere de suposiciones de modelo previas.

ESMO solo está implementado para calcular interacciones de tres SNPs, es decir, tiene una dimensión de tres o de k=3.

Aunque ESMO ya tenía la capacidad de aprovechar la computación paralela, solo utilizaba el dispositivo CPU. En este proyecto hemos ampliado esa capacidad,

pudiendo aprovechar toda la capacidad de computación paralela tanto de dispositivos CPUs, GPUs o incluso de forma heterogénea utilizando varios dispositivos CPU y GPU al mismo tiempo. Este proceso se ha llevado a cabo en tres fases: en la primera fase se ha traducido el lenguaje de código de Matlab (MathWorks, s.f.) (lenguage original en el que está escrito ESMO) a C++, consiguiendo una versión secuencial de ESMO en C++; en la segunda fase se ha implementado OpenCL (Khronos, s.f.) en C++ para aprovechar toda la potencia de computación paralela de los dispositivos CPU y GPU de forma individual a cada dispositivo; por último, en la tercera fase hemos utilizado OpenMP (OpenMP, s.f.) para lanzar los kernels OpenCL de cada dispositivo de forma simultánea, consiguiendo así que el programa sea compatible con una computación paralela heterogénea con varios dispositivos simultáneos, tanto CPUs, como GPUs.

4.1 Traducción de Matlab a C++

La traducción del lenguaje Matlab a C++ se ha intentado realizar de forma que el resultado sea lo más parecido posible en ambos programas, sin realizar refactorizaciones más allá de lo estrictamente necesario.

En esta fase, la complejidad residía en la forma de tratar las matrices de Matlab, y las diferencias con C++, y en los métodos o funciones nativas de Matlab, que no existían en C++ y se han tenido que crear.

En primer lugar, en el programa secuencial se decidió pasar a través de parámetros aquellos datos necesarios para la ejecución del programa. Estos datos son la ruta de la base de datos de dos dimensiones de SNPs de control y casos que se utilizará, el número de pacientes (número de filas en la base de datos), el número de SNPs más uno por la columna de estado de los pacientes (número de columnas en la base de datos) y el número de combinaciones que se ejecutarán por ventana. Esto es necesario si queremos ejecutar un número limitado de ventanas. Tras este paso necesitamos leer la base de datos. Para esto se ha creado un método que lee carácter por carácter y los guarda en una matriz de dos dimensiones de tipo "*char*". tras la lectura y almacenamiento de la base de datos en nuestro programa comenzamos la medición de tiempo utilizando la clase de C++ "*high_resolution_clock*" y llamamos a la función

"*exhausive_search2*" que es la función principal del programa. Esta función tiene como parámetros la base de datos en forma de matriz de dos dimensiones, la columna de estado de los pacientes (última columna de la matriz de la base de datos), la dimensión de la epistasia (k), el número de filas y el número de columnas de la base de datos, un vector con los factoriales precalculados (usado para ahorrar tiempo en el programa), una matriz de dos columnas donde se guardan los resultados de dos dimensiones de cada combinación de SNPs, una matriz de tres columnas donde se guardan las coordenadas de los tres SNPs de cada combinación de SNPs, el número de combinaciones por cada ventana, el vector donde se guardarán las combinaciones que se van a calcular en cada ventana y un entero que nos dice si hay tres o cuatro resultados no dominados en cada ventana. Esta función nos devuelve las combinaciones no dominadas con más probabilidad de interacción de cada ventana y las ventanas anteriores. Dentro de esta función se han tenido que implementar algunas funciones nativas de Matlab, como la función "*zeros*" para inicializar una matriz. también se ha tenido que cambiar la forma de inicializar matrices, ya que C++ no soporta funciones como inicializar una matriz a partir de otra, y la forma en la que se asignan los datos a las matrices, ya que Matlab es muy flexible en la gestión de datos con matrices, y nos permite asignar filas o columnas enteras sin necesidad de definir previamente el tamaño de las matrices, lo cual no se puede hacer en C++. En la versión secuencial de "*exhausive_search2*" también se han añadido un contador de combinaciones, un contador de ventanas, donde cada ventana calcula un número prefijado de combinaciones, y un limitador de ventanas, por si se desea que el programa pare al ejecutar un número concreto de ventanas, esto se ha realizado para poder paralelizar correctamente el programa utilizando OpenCL.

Dentro de "*exhausive_search2*" podemos encontrar dos funciones de ESMO, "*multiscore_obj2*", que nos proporciona los valores de las funciones objetivo de cada combinación. Esta función es llamada para todas las combinaciones posibles de 3 SNPs, y la función "*non_domiant_top_k*" que selecciona las tres o cuatro (dependiendo de si hay más de tres) mejores combinaciones teniendo en cuenta la puntuación de cada combinación obtenida en la función "*multiscore_obj2*". La función "*non_domiant_top_k*" se ha tenido que rehacer pensando en la posterior conversión a la computación paralela con OpenCL para adaptarlo al trabajo con ventanas, guardando las mejores soluciones de comparar la ventana actual con las mejores tres

o cuatro soluciones de las ventanas anteriores. Para poder conseguir esto, se ha ampliado el vector que contiene todas las soluciones que se van a comparar en 4 posiciones, donde se guardan las mejores soluciones de las ventanas anteriores utilizando la función también creada "*volcarBestSolutions*". En cuanto a la función "*multiscore_obj2*", llama a otra función, "*MutualInfo_improved*", que utiliza una función nativa de Matlab llamada "*accumarray*", que también se ha tenido que crear ya que en C++ no existe. Esta función es utilizada varias veces, tanto en el propio "*multiscore_obj2*" como en las funciones "*JointEntropy_accumarray*", "*JointEntropy_3loci_accumarray*" y "*MutualInfo_improved*". "*MutualInfo_improved*" es una función que utiliza la entropía H(Y) de Shannon para calcular la puntuación de cada emparejamiento de SNPs, considerándose las puntuaciones más bajas como alta probabilidad de epistasia, estas funciones pueden verse en el anexo al final del documento.

Para realizar todos estos cambios en dichas funciones, y, sobre todo, poder tratar las matrices como se tratan en Matlab, se han tenido que crear varios métodos auxiliares, entre ellos "*llenarVector*" que asigna el carácter pasado por parámetro a todos los elementos del vector, "*copiarMatriz3D*", que copia una matriz de tres dimensiones pasada por parámetro a otra, "*concatenarMatrices*", que concatena dos matrices pasadas por parámetro, "*separarColumnas*", que dado un vector de enteros con los identificadores de las columnas, construye una matriz con dichas columnas, "*rangoDeColumnas*", que crea una matriz con el rango de columnas especificado de otra que se le pasa por parámetro, "*sumarNaMatriz*", que suma un número a todos los elementos de la matriz, "*restarMatrices3D*", que resta dos matrices de tres dimensiones pasadas por parámetro y devuelve la matriz resultante, "*accumarray*" ya mencionado para tres y para 4 dimensiones, y, por último, "*minimoMatriz2D*", que coge el mínimo de los valores en ambas columnas de la matriz de puntuaciones de combinaciones de SNPs pasada por parámetro, y se guardan las dos menores puntuaciones junto con las dos posiciones de esas puntuaciones. Esta función es utilizada por la función "*non_domiant_top_k*" para la selección de las mejores combinaciones.

Para realizar el cálculo por ventanas primero se guardan todas las combinaciones para una ventana, y una vez que una ventana tiene todas sus combinaciones, se realizan los cálculos para esta ventana, se comparan los resultados con las ventanas anteriores,

se guardan los mejores resultados, y se pasa a la siguiente ventana donde se vuelve a hacer los mismo, hasta que se acaban todas las combinaciones posibles, si al acabarse las combinaciones no se ha llenado la última ventana, se ejecuta esta ventana para calcular las combinaciones que tenga.

Tras ejecutar la función "*exhausive_search2*" terminamos la medición de tiempo y mostramos el tiempo en milisegundos y las soluciones.

4.2 Implementación de OpenCL

La implementación de OpenCL nos permite paralelizar la versión secuencial del programa. El objetivo de esta implementación es poder aprovechar la potencia computacional de dispositivos CPU o GPU utilizando computación paralela.

Para realizar esta implementación se ha comprimido la base de datos de tal forma que un valor contenga los datos de varios SNPs y se han cambiado todas las variables posibles a tipo *char* para optimizar tanto el almacenamiento en memoria como el envío de información a los dispositivos GPUs. De esta forma, como los posibles valores a almacenar son 0, 1 y 2 (dos bits por dato), cada variable puede almacenar 4 datos diferentes. Después hemos creado una función de OpenCL llamada "*OpenCL_exhausive_search2*" a partir de la función secuencial "*exhausive_search2*". Esta función paraleliza la función "*exhausive_search2*" haciendo que cada work ítem del dispositivo utilizado, sea CPU o GPU, procese una serie de combinaciones de SNPs. Este número de combinaciones que procesa cada work ítem se obtiene de dividir el número de combinaciones por cada ventana que se ha dado, entre el número de work ítems definido para el dispositivo. El algoritmo para cada ventana es el siguiente:

```
INICIO
1   #Ejecución de una ventana
2   Para i desde 0 al número de SNPs -2
3       Para j desde i + 1 al número de SNPs -1
4           Para k desde j + 1 al número de SNPs
5               Guardar los 3 SNPs de la combinación en vector de combinaciones
6               Sumar 1 al contador de combinaciones
7               Si contador de combinaciones es igual al tamaño de la ventana
8                   Se definen los objetos OpenCL del vector de combinaciones y el contador de
                    combinaciones en memoria
9                   Pasamos estos objetos definidos como parámetros al kernel
10                  Ejecutamos el Kernel
11                  Esperamos a que finalicen todos los work ítems
```

```
12                  Leemos los buffers de salida All_score (los valores de todas las combinaciones de la
                    ventana), y All_pos (las coordenadas de todas las combinaciones de la ventana)
13                  Obtenemos las mejores soluciones llamando a non_dominant_top_k con los datos de
                    All_score, All_pos y las mejores soluciones anteriores
14                  Reiniciamos el contador de combinaciones a cero
15              Fin Si
16          Fin Para
17      Fin Para
18  Fin Para
FIN
```

Cada ventana devuelve los tres (o cuatro si los hay) mejores resultados de epistasia en dos dimensiones, y también se comparan estos resultados con los mejores obtenidos en las ventanas anteriores para quedarnos en cada ventana completada con los mejores resultados de todas las ventanas anteriores. De esta forma, al finalizar el programa obtenemos los mejores resultados de todas las combinaciones de SNPs. Para poder hacer esto, se ha implementado un fichero llamado "programa.cl" que contiene los kernels OpenCL para GPU y CPU que son utilizados dentro de la función "*OpenCL_exhausive_search2*" y a los que, pasándole por parámetro el vector con todas las combinaciones que se van a calcular en esa ventana, realiza los cálculos de forma paralela utilizando los work ítems indicados:

```
INICIO
1   #Kernel de OpenCL, ejecución para cada work ítem
2   Se inicializan las variables necesarias
3   Se obtiene el identificador global del work item
4   Se obtiene el número total de work items
5   Se calcula el número de combinaciones que debe ejecutar cada work item
6   Para cada combinación desde 0 hasta el número de combinaciones que cada work item debe ejecutar
    (En este bucle se encuentra la diferencia en el acceso a memoria entre la GPU y la CPU, se verá con
    más detenimiento en el apartado 4.2.2)
7       Se obtiene del vector de combinaciones los tres SNPs de la combinación
8       Se forma una matriz de 4 columnas con las 3 columnas de los SNPs y la columna de estado de
        la matriz de datos
9       Se llama a multiscore_obj2 pasándole esta matriz de 4 columnas para obtener los resultados de
        esta combinación
10      Se guardan los resultados en la posición correspondiente a la combinación en All_score (los
        valores de todas las combinaciones de la ventana) y All_pos (las coordenadas de todas las
        combinaciones de la ventana)
11  Fin Para
FIN
```

Hay que tener en cuenta que en el kernel de OpenCL no se pueden definir vectores con una variable que defina el espacio, o lo que es lo mismo, solo se puede definir vectores con un espacio predefinido, por lo que aquellos vectores que se necesitan dentro del kernel pero que su tamaño depende de otros valores como parámetros, el

tamaño de la base de datos, etc… se tienen que definir fuera del kernel y pasárselo a éste como parámetro. Por último, resaltar que los modelos de almacenamiento de los kernel de OpenCL para GPU y CPU son distintos para adaptarse a las características de cada dispositivo, como se verá con más detalle en el apartado 4.2.2.

En cuanto a la función "*OpenCL_exhausive_search2*", tiene los mismos parámetros de entrada que en la versión secuencial, y se puede decir que se divide en dos partes, la primera parte inicializa todas las variables necesarias para el programa así como todos los objetos necesarios para lanzar un kernel OpenCL: obtiene y selecciona la plataforma en la que se ejecutará el kernel, obtiene y selecciona el dispositivo donde se ejecutará el kernel dentro de la plataforma seleccionada, carga el código OpenCL contenido en "programa.cl" (funciones y kernels), lo compila en el dispositivo seleccionado, carga el número de work ítems que ejecutarán el kernel, crea los objetos de memoria donde se asignarán las variables que se van a pasar por parámetro al kernel, transfiere las variables al dispositivo a través de estos objetos de memoria y enlaza los objetos de memoria con los parámetros del kernel. Las variables del kernel pueden ser definidas en las memorias privada, constante, local, o global. Las variables que no se necesitan compartir entre work ítems, como el número de SNPs, la dimensión de la epistasia, el número de filas y columnas de la base de datos y el número de combinaciones a realizar, han sido declaradas en la memoria privada, ya que tiene el acceso más rápido; sin embargo, el dispositivo tiene una memoria privada muy pequeña, por lo que hay otras variables, como la columna de estado o el vector con los factoriales precalculados, que se ha decido almacenar en la memoria constante, ya que no cabían en la memoria privada, o la base de datos que, a pesar de estar comprimida, se ha tendido que declarar en la memoria global porque es demasiado grande para declararla como local. Por otra parte, las variables que se comparten entre work ítems, como las matrices donde se guardan las soluciones de cada combinación, o las variables utilizadas por el programa para calcular las soluciones de cada combinación, también están definidas en la memoria global. Estas últimas, hay que tener en cuenta que al calcularse las combinaciones de forma paralela por cada work ítem, los vectores necesarios para el cálculo multiplican su tamaño por el número total de work ítems, ya que cada work ítem necesita su propio espacio dentro de cada uno de estos vectores para el cálculo de las combinaciones.

En segundo lugar esta función guarda en un vector las combinaciones que se van a calcular de forma paralela en cada ventana (de la misma forma que en la versión secuencial), pasa como parámetro al kernel este vector de combinaciones, y ejecuta el kernel, que devuelve una vez completado las mejores soluciones de la ventana, y esta se compara con las mejores soluciones de las ventanas anteriores, siguiendo la misma metodología que en la versión secuencial. Después se vuelve a obtener otro vector de combinaciones y se vuelve a ejecutar el kernel hasta que no quedan más combinaciones.

El tiempo de ejecución se mide desde que comienza la función "*OpenCL_exhausive_search2*" hasta que acaba. Al final de esta función se muestran los resultados de las mejores combinaciones y el tiempo que el programa ha necesitado, de la misma forma que en la versión secuencial.

4.2.2. Organización de Memoria para GPU y CPU

Para la gestión de memoria en la GPU se ha implementado un modelo basado en coalescencia, es decir, para aprovechar el ancho de banda de la GPU, hacemos que los cores de la GPU del mismo wrap accedan a la vez a posiciones de memoria consecutiva. Para esto, tenemos que guardar en el vector de combinaciones en el host todas las posiciones de los SNPs de cada coordenada de forma consecutiva, obteniendo así un primer bloque con todas las posiciones de los primeros SNPs de cada combinación, después un bloque con todas las posiciones de la segunda coordenada de la combinación, y por último un bloque con las posiciones de la tercera coordenada de la combinación de SNPs, y en el kernel, cuando accedemos a este vector con los tres bloques, hacer que cada work item acceda a cada SNP para cada combinación, de esta forma, todos los datos se leen al mismo tiempo, como podemos ver en la figura 4.1.

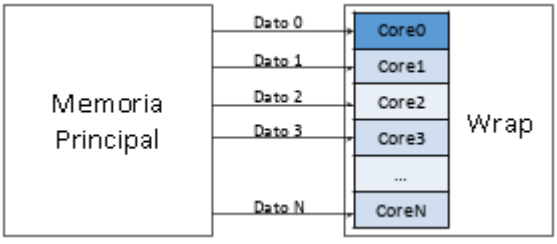

Figura 4.1 Acceso de la GPU a los datos en memoria.

Por otro lado, para la gestión de memoria en la CPU, se ha implementado un modelo que aprovecha la caché, donde se leen los datos de forma consecutiva en los bloques de memoria, para esto se necesita que las combinaciones se guarden en el vector de combinaciones de forma consecutiva en el host, es decir, que las tres posiciones de los tres SNPs de cada combinación se almacenen en el vector una seguida de otra, con los tres SNPs que forman la combinación juntos, y que luego en el kernel se cada work item acceda a los SNPs de la combinación también de forma consecutiva en memoria.

4.3 Implementación de OpenMP

El último paso que hemos realizado ha sido la implementación de OpenMP para poder paralelizar la ejecución de varios dispositivos al mismo tiempo, consiguiendo así toda la potencia computacional de varios dispositivos simultáneos.

Para hacer esto hemos utilizado la directiva de OpenMP "*pragma omp parallel*" con un "*thread*" o hilo por cada dispositivo que queremos usar de forma paralela, y las directivas de OpenMP "*pragma omp sections*" para crear un bloque de secciones donde cada sección lo va a ejecutar un hilo distinto, y la directiva "*pragma omp section*" para ejecutar la función "*OpenCL_exhausive_search2*" vista en el apartado anterior en cada dispositivo de forma simultánea.

Hemos añadido un parámetro de tipo *char* a la función "*OpenCL_exhausive_search2*", este parámetro puede ser 'G', para hacer referencuia a que se tiene que ejecutar el kernel de GPU, o 'C', para hacer referencia a que se tiene que ejecutar el kernel de tipo CPU, esto se debe a la diferencia del acceso a memoria del vector combinaciones del kernel que existe para la GPU y la CPU (recordemos que cada dispositivo usa un kernel diferente).

También se han agregado otros parámetros: dos parámetros para definir la plataforma y el dispositivo que ejecutará el kernel en cada llamada en OpenMP, tres enteros que se compartirán entre todos los hilos o "*threads*" pero que estarán siempre dentro de una sección crítica de OpenMP y que se utilizarán para obtener todas las combinaciones de los vectores de combinaciones dentro de "*OpenCL_exhausive_search2*", de forma que dos hilos no puedan calcular la misma combinación de SNPs, el número total de ventanas, que también es compartido por los hilos de OpenMP y que nos dice cuántas ventanas se han hecho entre todos los hilos

de OpenMP, y un *string* que nos permitirá saber qué dispositivo está almacenando el resultado en el archivo de salida.

Cada dispositivo almacenará sus tres o cuatro mejores soluciones de todas las combinaciones procesadas, así como su tiempo de ejecución, y el número de ventanas que ha ejecutado.

4.4 Selección de Pruebas a Realizar y Descripción de los Datos Escogidos

En primer lugar, se ha realizado una prueba de la versión secuencial con una base de datos de 31341 SNPs y 146 pacientes con un tamaño de ventana de 50000 (el tamaño de ventana no afecta en absoluto, ya que fue un paso intermedio para implementar posteriormente OpenCL) y procesando un millón de combinaciones para extrapolar los resultados al tamaño total de combinaciones de la base de datos (5,13e12 combinaciones para k=3). Esto se debe a que era inviable realizar estas pruebas de forma completa por el tiempo que tarda el programa, el cual se ha estimado que tardaría en torno a 638 días. Esta prueba se ha ejecutado cinco veces para asegurar la veracidad de ésta, y se ha tomado la media como resultado.

En segundo lugar, se han realizado un estudio paramétrico de la versión de OpenCL para el dispositivo GPU Titan X y el dispositivo CPU para encontrar la mejor combinación entre el número de work ítems y la carga de trabajo que realizarán cada uno. Estas pruebas se han realizado con una base de datos de 1000 SNPs y 4000 pacientes.

En cuanto a la GPU Titan X, teniendo en cuenta los 3072 cores de la tarjeta gráfica, se han hecho pruebas con cuatro números distintos de work ítems, 3072, 6144, 9216 y 12288, que corresponden al número de cores multiplicado por uno, dos, tres, y cuatro respectivamente, con el objetivo obtener el número de work ítems óptimo para este dispositivo. Con cada valor para el número de work ítems se han hecho distintas pruebas a su vez con el número de combinaciones para cada work ítem, seleccionando 10, 100, 1000 y 10000 combinaciones por work ítem, para intentar obtener, además del número óptimo de work ítems del dispositivo, el número óptimo de combinaciones que debe resolver cada work ítem. De esta forma en función de las dos variables (número de work ítems y número de combinaciones por work ítem) se han realizado

dieciséis pruebas distintas, y cada prueba se ha ejecutado once veces para asegurar la veracidad de la prueba, sacando al final una media de las once ejecuciones.

En cuanto a la CPU, y tomando como referencia que tiene 16 cores, se han probado tres números distintos de work ítems para encontrar el más óptimo, 16 work ítems, 32 work ítems, y 64 work ítems, y, al igual que en el caso de la GPU, también se ha probado con los mismos cuatro números distintos de combinaciones por work ítem, teniendo en este caso doce pruebas distintas, y también como en el caso de la GPU, cada prueba se ha ejecutado once veces para asegurar la veracidad de la misma, sacando una media de todas las ejecuciones.

En tercer lugar, se han realizado pruebas con la base de datos de 31341 SNPs y 146 pacientes a la CPU y a la GPU, estas pruebas han sido realizadas limitando la ejecución del programa a 5000 ventanas de 61.440.000 combinaciones cada una (recordemos que esto ha sido necesario por el elevado tiempo de ejecución que hacía inviable realizar todas las pruebas completas), ejecutándose la CPU con 64 work ítems y la GPU con 6144 work ítems. Estos datos salen de los resultados del estudio anterior, donde se concluye que estos números de work ítems son los más óptimos para estos dispositivos y que el tamaño de ventana no tiene un gran impacto (lo veremos con detalle en el apartado siguiente). Estas pruebas se han ejecutado cinco veces para asegurar la veracidad de las mismas.

Indicar que las dos bases de datos utilizadas son bases de datos simuladas que han sido generadas mediante el software GAMETES 2.0 y que pueden descargarse de http://arco.unex.es/granado/SNP/

Por último, se han realizado pruebas de la versión completa de OpenMP+OpenCL utilizando los dispositivos CPU, GPU Titan X y GPU K40m de forma simultánea. Estas pruebas se han realizado igual que las anteriores, es decir, limitando la ejecución del programa a 5000 ventanas de 61.440.000 combinaciones cada una, y utilizando 64 work ítems para la CPU, 6144 para la GPU Titan X, y 5760 para la GPU K40m.

En los siguientes apartados se mostrarán los resultados de todas las pruebas realizadas y se discutirán las conclusiones obtenidas.

5

RESULTADOS Y DISCUSIÓN

Para realizar los experimentos se ha utilizado un nodo con dos procesadores Intel® Xeon® CPU E5-2630 v3 a 2.40 GHz con 8 núcleos por procesador y Hyperthreading (permitiendo en total 32 hilos en paralelo), 82 GB de RAM, una GPU Nvidia GeForce GTX TITAN X con 3072 núcleos CUDA a 1076 MHz y 12213 MB de RAM a 3505 MHz y una GPU Nvidia Tesla K40m con 2880 núcleos CUDA a 750 MHz y 11441 MB de RAM a 3004 MHz. El nodo tiene instalado un sistema operativo Ubuntu 18.04 LTS, gcc 8.3.0 y OpenCL 1.2, que es el soportado por las GPUs utilizadas. Es importante destacar que no se muestran resultados epistáticos concretos ya que, al ser una búsqueda exhaustiva, se prueban todas las combinaciones y el resultado es siempre el mismo. Además, al ser bases de datos generadas, estos resultados no tienen un interés biológico por sí mismos. Eso sí, se ha comprobado que todas las versiones dan el mismo resultado que, además, también coincide con el generado por ESMO.

5.1 Optimización de Parámetros para los Dispositivos CPU y GPU

Antes de comenzar la fase de experimentación, se realizó un estudio paramétrico usando la CPU y la GPU Titan X. Estas pruebas se realizaron utilizando una base de datos de 1000 SNPs y 4000 pacientes, y las variables modificadas fueron el número de work ítems y el número de combinaciones calculadas por cada work ítem modificando el tamaño de ventana. En la figura 5.1 se muestran los resultados obtenidos.

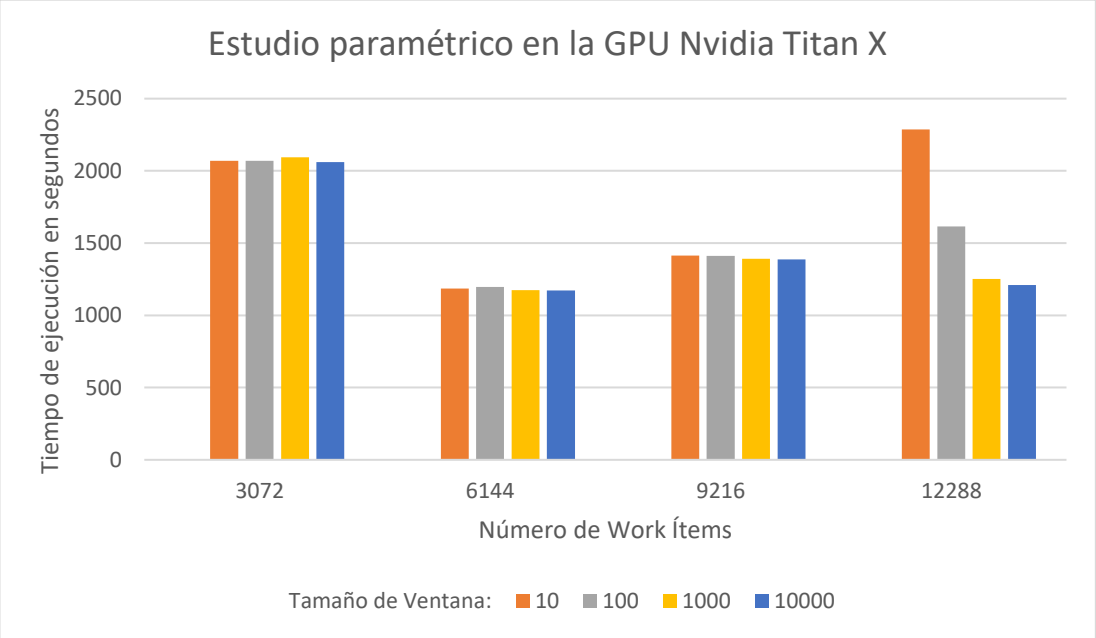

Figura 5.1 – Resultados de la ejecución en el dispositivo GPU Nvidia Titan X en una base de datos de 1000 SNPs y 4000 pacientes.

La gráfica de la figura 5.1 representa los resultados de las pruebas realizadas con la base de datos de 1000 SNPs y 4000 pacientes en el dispositivo Nvidia Titan X. Cabe destacar que cada prueba se ha ejecutado once veces, y que el resultado es la media de todas las ejecuciones. En el eje Y se representa el tiempo que ha tardado el programa en completarse en segundos, en el eje X se representa el número de work ítems utilizados en cada prueba, y cada barra representa el número de combinaciones calculadas por cada work ítem. De estos resultados podemos sacar varias conclusiones: en primer lugar, podemos observar que el número de combinaciones realizadas por cada work ítem, o lo que es lo mismo, el tamaño de ventana del programa, en general

no es un factor determinante, aunque vemos que en las pruebas con 12288 work ítems sí varía, debido a que al ser un tamaño de ventana tan pequeño y una relación tan alta de cores CUDA y work ítems (1 a 4) hace haya muchos work ítems que tienen que esperar a otros. En cualquier caso, podemos observar que con un tamaño de ventana de 10000 combinaciones se consiguen los mejores resultados en todas las pruebas, por lo que es este valor el que vamos a utilizar para realizar la experimentación.

En segundo lugar, vemos que el número de work ítems sí afecta sustancialmente al rendimiento. Concretamente, vemos que la opción más adecuada para el dispositivo GPU Nvidia Titan X es 6144, que corresponde al doble de su número de cores. Esta observación la confirmamos con el dispositivo GPU Nvidia Tesla K40m, que tiene 2880 núcleos, y donde obtenemos la máxima eficiencia utilizando OpenCL con 5760 work ítems. Estos números de work ítems serán utilizados para la prueba final en estos dispositivos.

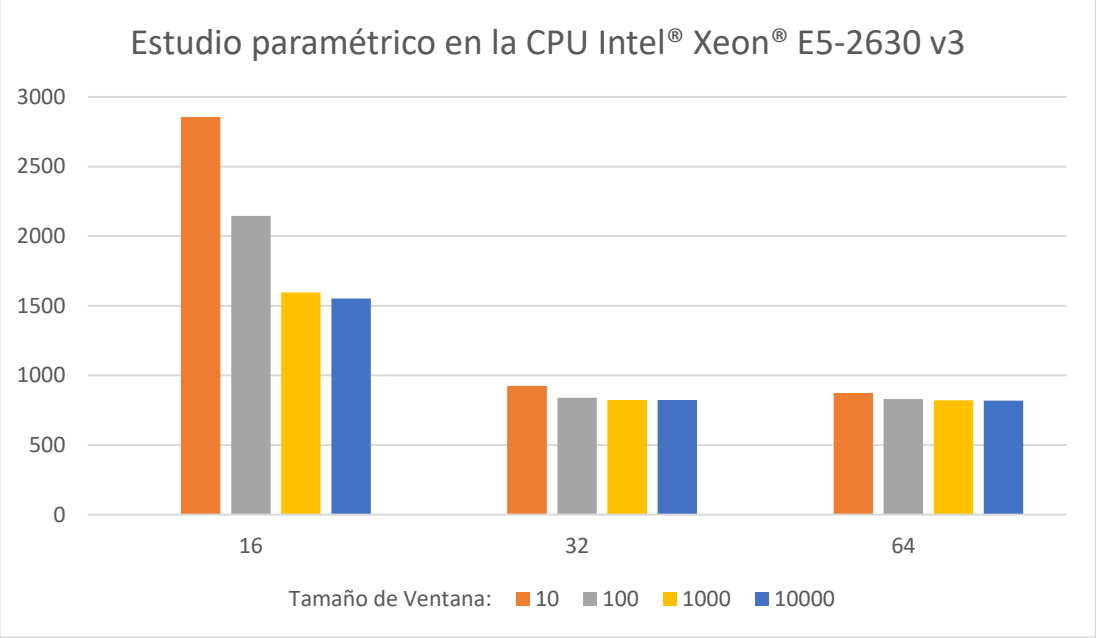

Figura 5.2 - Resultados de la ejecución en el dispositivo CPU 2xIntel Xeon E5-2630 v3 en una base de datos de 1000 SNPs y 4000 pacientes.

El mismo proceso se ha realizado con la CPU. La figura 5.2 presenta los resultados del estudio paramétrico realizado con la base de datos de 1000 SNPs y 4000 pacientes en el dispositivo CPU 2xIntel Xeon E5-2630 v3 de 16 núcleos y 32 hilos a 2,40 GHz. La representación es la misma que en la anterior, solo que en este caso se ha probado con 16, 32, y 64 work ítems.

Al igual que en el caso anterior, vemos que el tamaño de la ventana no tiene un gran impacto en el rendimiento, excepto en el caso de los 16 work ítems, debido a que, al lanzar el mismo número de work ítems que de hilos del CPU, se desaprovecha la autovectorización de los datos, y, al haber pocos datos en cada ventana, se produce mucha desincronización entre kernels.

También podemos observar que, aunque no hay una gran diferencia entre ejecutar el programa con 32 o 64 work ítems, los tiempos son ligeramente mejores con 64 work ítems, por lo que se utilizará este número de work ítems para la experimentación final en la CPU.

5.2 Experimentación

La experimentación se ha realizado utilizando una base de datos de tamaño más cercano a las utilizadas en la realidad. Concretamente esta base de datos contiene 31341 SNPs y 146 pacientes. Además, se han utilizado los parámetros obtenidos en el ajuste paramétrico descrito en el apartado anterior. Es importante destacar que, debido a los elevados tiempos de ejecución (más 9 días en el mejor de los casos y casi 2 años en el peor, como veremos a continuación) y que teníamos que realizar varias repeticiones por ejecución, era inviable hacer ejecuciones completas, por lo que se optó por procesar una porción suficientemente grande del número de combinaciones, medir el tiempo requerido para dicha porción y estimar el tiempo necesario para la ejecución final. En el caso de la versión secuencial se ha ejecutado el programa con 1 millón de combinaciones, mientras en el caso de las versiones de OpenCL se han realizado las pruebas ejecutando el programa con 5000 ventanas de 61.440.000 de combinaciones (307.200.000.000 combinaciones), y después se han extrapolado los tiempos al número total de combinaciones (5.130.335.012.110). Esto significa que la ejecución cubre algo más de 1/16 parte de las ejecuciones paralelas.

Para las versiones OpenCL se han realizado tres experimentos, en el primero se ha ejecutado el programa únicamente sobre la CPU, en el segundo se ha ejecutado el programa únicamente sobre la GPU Titan X, y en el tercer se ha ejecutado la versión final del programa con OpenMP+OpenCL utilizando simultáneamente los dispositivos CPU, GPU Titan X y GPU K40m.

Podemos ver los resultados finales de los cuatro experimentos en la figura 5.3.

Figura 5.3 – Tiempos de ejecución de las versiones secuencial y paralelas con una base de datos de 31341 SNPs y 146 pacientes.

En la gráfica de la figura 5.3 podemos ver que hay una diferencia muy grande entre la versión secuencial y las versiones paralelas implementadas con OpenCL, siendo la versión CPU la que tiene peores tiempos. En concreto la versión secuencial ha necesitado 15.313,28 horas (1,75 años), mientras que las versiones paralelas reducen significativamente el tiempo a 423,69 horas (17,65 días) en la versión CPU, 394,24 horas (16,43 días) en la GPU Titan X y 218,13 horas (9,01 días) en la versión heterogénea, lo que nos da SpeedUps de 36,14, 38,94 y 70,39 respectivamente.

Una vez que hemos comprobado que la paralelización reduce significativamente el tiempo de ejecución requerido, vamos a analizar en detalle los tiempos obtenidos por cada una de las versiones paralelas (figura 5.4).

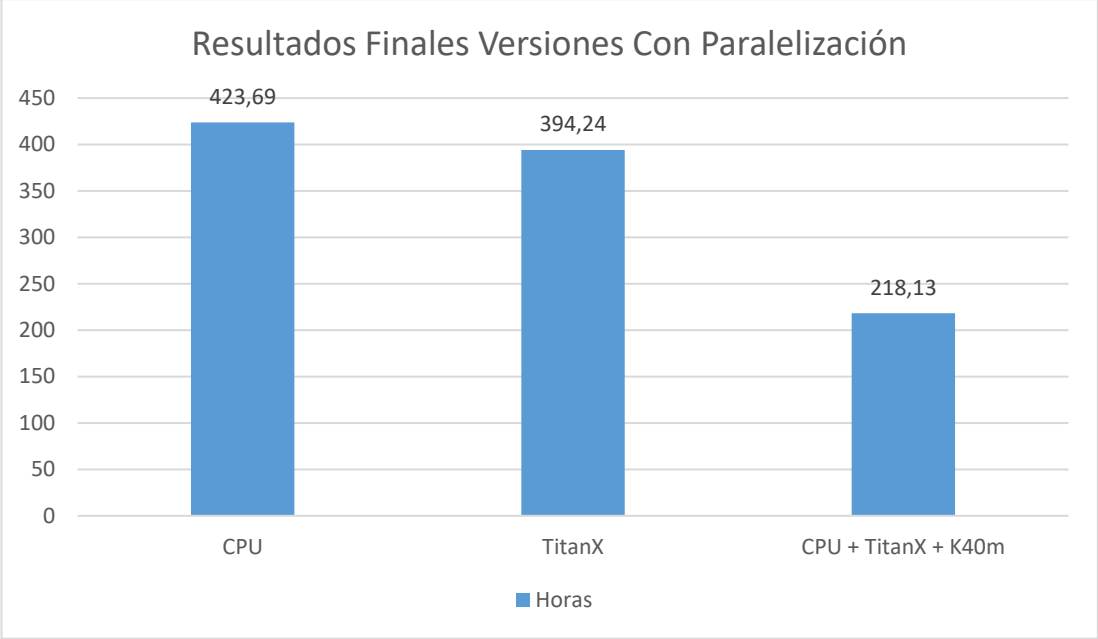

Figura 5.4 – Tiempos de ejecución de las versiones paralelas con una base de datos de 31341 SNPs y 146 pacientes.

En la gráfica de la figura 5.4 podemos ver que se obtienen mejores tiempos paralelizando con OpenCL en el dispositivo GPU (394,24 horas) que en el dispositivo CPU (423,69 horas), aunque la diferencia es pequeña, ya que se consigue en el dispositivo GPU TitanX un SpeedUp de 1,075. Esta ganancia temporal no es muy significativa teniendo en cuenta la diferencia entre los dispositivos y, tras mucho estudio, hemos llegado a la conclusión de que el problema radica en la forma en que se procesan las funciones objetivo. Estas funciones requieren que cada work ítem realice accesos aleatorios a la base de datos, lo que provoca que se pierda la coalescencia en memoria de la GPU. En la sección de metodología hemos mostrado cómo hemos tratado de reducir este hecho, pero ha sido imposible eliminarlo por completo sin plantear una completa reestructuración de la implementación (desafortunadamente, este cambio implicaría prácticamente empezar el TFG desde el principio, por lo que se hacía inabarcable). Además, hay que tener en cuenta que el compilador de OpenCL de Inter realiza autovectorización, mientras que el de NVIDIA no. Aun así, la versión GPU consigue reducir considerablemente el tiempo de ejecución secuencial y mejorar el tiempo CPU. Donde si vemos una diferencia notable es en la versión heterogénea, donde se combinan OpenMP para gestionar los dispositivos de forma simultánea y OpenCL para la paralelización del código. En este

caso, al utilizar la potencia computacional combinada de la CPU, la GPU TitanX y la GPU K40m de forma simultánea, se obtiene el mejor tiempo de ejecución (218,13 horas), consiguiendo un Speedup de 1,94 respecto a la versión de la CPU, y un Speedup de 1,8 respecto a la versión de la GPU TitanX. Indicar que, tras estudiar los datos procesados por cada dispositivo, podemos indicar que la mayor carga de trabajo la realiza la GPU Titan X, seguida de la CPU y, por último, de la GPU k40. Creemos que la diferencia de procesamiento entre la Titan X y la K40 radica en que la Titan X tiene más núcleos CUDA que la K40 y que, además, trabajan a mayor frecuencia.

6

CONCLUSIONES

6.1 Conclusiones del Estudio

Del estudio realizado podemos concluir que, en primer lugar, la versión paralela de un solo dispositivo aumenta enormemente el rendimiento en tiempo del programa (más de diez veces en la versión de CPU Paralela frente a la versión CPU secuencial). En segundo lugar, vemos que en las versiones que implementan paralelismo con OpenCL, el número de combinaciones realizadas por cada work ítem (tamaño de ventana) no tiene gran importancia, a diferencia del número de work ítems lanzados, que si se pueden observar diferencias notables. Además, podemos observar también que en el caso de los dispositivos GPU el número de work ítems más óptimo equivale al doble de cores del dispositivo, y en el caso de la CPU, aunque el número óptimo de work ítems equivale al doble de núcleos del procesador, o lo que es lo mismo, coincide con el número de hilos paralelos (cores+*hyperthreading*) que permite el procesador. Vemos que no hay una gran diferencia entre este número de work ítems y el

equivalente al doble de hilos, pero si podemos observar una diferencia notable entre lanzar el mismo número de work ítems que el número de cores, y lanzar el doble de work ítems que el número de cores ya que se pierde la ventaja ofrecida por el *hyperthreading*. Por último, podemos observar que al ejecutar la versión implementada usando OpenMP y OpenCL, utilizando la computación paralela heterogénea, mejora de forma proporcional al número y potencia de los dispositivos por lo que se puede concluir que esta versión ha sido exitosa, sin embargo, esta mejora no es 1:1, es decir, si tenemos dos dispositivos con la misma potencia computacional, el programa en realidad no tarda la mitad de lo que tardaría con un solo dispositivo, sino que tardaría algo más de la mitad, debido a algunas características de la versión paralela heterogénea como el envío de datos a los distintos dispositivos, partes de la ejecución que son estrictamente secuenciales, etc.

Además, podemos concluir que, aunque el uso de GPUs reduce considerablemente el tiempo de ejecución con respecto a la versión secuencial, el problema tratado no es óptimo para la implementación en dispositivos GPU, donde los diferentes núcleos acceden de forma simultánea a posiciones consecutivas de memoria (coalescencia). En el problema tratado en este trabajo, hemos visto que los diferentes núcleos CUDA deben acceder a posiciones no consecutivas en memoria al calcular los valores de las funciones objetivo, haciendo que se pierda la coalescencia de memoria y, por tanto, reduciendo el rendimiento del código GPU. Como posible trabajo futuro podría plantearse invertir la orientación de los datos de tal forma que los diferentes work ítems no calculen cada uno las funciones objetivo de diferentes combinaciones de SNPs sino que trabajen en conjunto para calcular las funciones objetivo de una única combinación de SNPs.

6.2 Conclusiones Personales

Desarrollar este proyecto de software, junto con el estudio asociado, ha sido muy didáctico por diversas razones. He podido aprender lo básico del lenguaje de programación Matlab, y profundizar y perfeccionar mis habilidades en el lenguaje de programación en C y en C++; también me ha permitido explorar la rama informática de la computación paralela, a través de los lenguajes OpenCL y OpenMP, así como su aplicación real en el estudio de la medicina y la genética. También me ha resultado

muy interesante poder trabajar de esta manera con los dispositivos CPU y GPU aprovechando todos los recursos de la forma más optimizada posible. Por otra parte, el estudio del arte sobre el problema de la epistasia me ha permitido, a través de los distintos artículos, aprender sobre diversos conceptos de biología, genética, estadística, y, por supuesto, algoritmos informáticos. Por todo esto, aunque también he tenido momentos de alguna dificultad, me ha resultado didáctico, interesante, y también constructivo desde el punto de vista de mi futura carrera profesional.

GLOSARIO

1. **Agente de aprendizaje de refuerzo**: El aprendizaje por refuerzo es un área del aprendizaje automático inspirada en la psicología conductista, cuya ocupación es determinar qué acciones debe escoger un agente de software en un entorno dado con el fin de maximizar alguna noción de "recompensa" o premio acumulado.
2. **Algoritmo de agrupación K-means**: Es un método de agrupamiento, que tiene como objetivo la partición de un conjunto de n observaciones en k grupos en el que cada observación pertenece al grupo cuyo valor medio es más cercano.
3. **Algoritmo de búsqueda armónica**: El algoritmo de búsqueda armónica (Harmony Search Algorithm) es un algoritmo meta heurístico que basa su funcionamiento en el proceso de la improvisación musical. El algoritmo de búsqueda armónica ha sido aplicado en infinidad de problemas de optimización, mostrando su eficiencia frente a otras metaheurísticas y otras técnicas matemáticas de optimización.
4. **Algoritmo de mini-lotes**: Es un algoritmo de entrenamiento para un agente de inteligencia artificial, el entrenamiento estocástico es realizar un entrenamiento en un ejemplo seleccionado al azar, mientras que el entrenamiento en mini lotes es el entrenamiento en una parte de los ejemplos generales. Los tamaños de mini lotes pueden variar según el tamaño de los datos.
5. **Algoritmo top-k**: Es un algoritmo que entresaca los mejores elementos de un conjunto de datos, estos pueden ser lo que tengan más puntuación, los que aparezcan con más frecuencia, etc…
6. **Aproximación de superposición de Kirkwood (KSA)**: La aproximación de superposición de Kirkwood establece que el potencial de la fuerza media de,

digamos, tres partículas, puede aproximarse por la suma del potencial de las fuerzas medias de pares de partículas.

7. **Biomarcadores**: Un biomarcador genético hace referencia a la fracción de ADN que nos indica una característica diferencial entre dos individuos, pudiendo así realizar un cribado gracias a éste. También puede ser una secuencia de ADN que causa una enfermedad en concreto o que está relacionada con la susceptibilidad a padecerla.
8. **Curvas de características operativas del receptor (ROC)**: La curva ROC es una expresión gráfica que nos proporciona una expresión del potencial diagnóstico de un marcador independiente de la población de pacientes y puede utilizarse para comparar uno o más marcadores.
9. **Capacidad de generalización**: Se denomina generalización a un proceso mediante el cual se establece una conclusión de índole universal desde una observación u observaciones particulares.
10. **Coeficiente de Gini**: El coeficiente de Gini es una medida de la desigualdad ideada por el estadístico italiano Corrado Gini. El coeficiente de Gini es un número entre 0 y 1, en donde 0 se corresponde con la perfecta igualdad (todos tienen los mismos ingresos) y donde el valor 1 se corresponde con la perfecta desigualdad.
11. **Desviación del modelo aditivo lineal**: Se refiere a la desviación, es decir, la medida de la diferencia entre el valor observado de una variable y algún otro valor, a menudo la media de esa variable, que existe en el modelo lineal aditivo, un modelo aditivo es un modelo de datos en el cual los efectos de factores individuales son diferenciados y agregados de manera conjunta para modelar los datos.
12. **Falsos positivos**: Se denomina como falso positivo a aquel error en el cual se incurre cuando el investigador rechaza la llamada hipótesis nula (aquella hipótesis que se crea con la misión de rechazar y por tanto de apoyar una hipótesis alternativa), siendo en efecto y por el contrario válida la misma en la población que se estudia.

13. **Genoma**: El genoma es el conjunto de genes contenidos en los cromosomas, lo que puede interpretarse como la totalidad del material genético que posee un organismo o una especie en particular.

14. **Locus**: Un locus es una posición fija en un cromosoma, que determina la posición de un gen o de un marcador (marcador genético). El plural de locus es loci.

15. **Método de corrección de prueba múltiple de Bonferroni**: La corrección de Bonferroni es uno de los varios métodos utilizados para contrarrestar el problema de las comparaciones múltiples. Las comparaciones múltiples, multiplicidad o problema de múltiples pruebas se produce cuando uno considera un conjunto de inferencias estadísticas simultáneamente o infiere un subconjunto de parámetros seleccionados en base a los valores observados.

16. **Método de validación cruzada**: La validación cruzada es una técnica utilizada para evaluar los resultados de un análisis estadístico y garantizar que son independientes de la partición entre datos de entrenamiento y prueba.

17. **Modelo estadístico G-test**: las pruebas G-test son pruebas de significación estadística de razón de verosimilitud o de máxima verosimilitud que se utilizan cada vez más en situaciones en las que anteriormente se recomendaban las pruebas chi-squared o $\chi 2$ test.

18. **OpenCL**: Es un estándar abierto de programación paralela, desarrollado por el consorcio Khronos Group en el año 2008. Este estándar permite crear aplicaciones que pueden ejecutarse paralelamente en procesadores gráficos (GPU) o centrales (CPU), distintos en su arquitectura y ubicados en un sistema heterogéneo

19. **OpenMP**: Es una interfaz de programación de aplicaciones (API) para la programación multihilo en memoria compartida en múltiples plataformas. Permite añadir concurrencia a los programas escritos en C, C++ y Fortran sobre la base del modelo de ejecución fork-join.

20. **Proceso de decisión de Markov**: Es un proceso de selección utilizado en inteligencia artificial basado en la propiedad de Markov, donde el estado anterior y la última acción realizada son suficientes para describir el estado

actual y el refuerzo recibido, por lo tanto, la acción a realizar solo depende del estado actual.

21. **Reproducibilidad**: La reproducibilidad es la capacidad de un ensayo o experimento de ser reproducido o replicado por otros, en particular, por la comunidad científica.
22. **Sensibilidad y especificidad**: La sensibilidad es la probabilidad de clasificar correctamente a un individuo enfermo, es decir, la probabilidad de que para un sujeto enfermo se obtenga en una prueba diagnóstica un resultado positivo.
23. **SNP (*Single Nucleotide Polymorphism*)**: Un polimorfismo de un solo nucleótido o SNP es una variación en la secuencia de ADN que afecta a una sola base (adenina (A), timina (T), citosina (C) o guanina (G)) de una secuencia del genoma.
24. **Tabla bidimensional de algoritmo tabú**: La búsqueda tabú es un método de optimización matemática, perteneciente a la clase de técnicas de búsqueda local. La búsqueda tabú aumenta el rendimiento del método de búsqueda local mediante el uso de estructuras de memoria: una vez que una potencial solución es determinada, se la marca como "tabú" de modo que el algoritmo no vuelva a visitar esa posible solución. En este caso, se utiliza una estructura de tabla bidimensional.
25. **Tabla de contingencia**: Las tablas de contingencia se emplean para registrar y analizar la asociación entre dos o más variables, habitualmente de naturaleza cualitativa (nominales u ordinales).
26. **Técnica de información mutua (*mutual entropy*)**: La información mutua o transinformación de dos variables aleatorias es una cantidad que mide la dependencia mutua de las dos variables, es decir, mide la reducción de la incertidumbre (entropía) de una variable aleatoria, X, debido al conocimiento del valor de otra variable aleatoria Y.

BIBLIOGRAFÍA

Aflakparast, M. y otros, 2014. Cuckoo search epistasis: a new method for exploring significant genetic interactions, DOI: 10.1038/hdy.2014.4. 112(666–674).

Goudey, B. y otros, 2013. GWIS - model-free, fast and exhaustive search for epistatic interactions in case-control GWAS, DOI: 10.1186/1471-2164-14-S3-S10.

Gou, J. y otros, 2014. Stability SCAD: a powerful approach to detect interactions in large-scale genomic study, DOI: 10.1186/1471-2105-15-62.

Han, B. & Chen, X.-w., 2011. bNEAT: a Bayesian network method for detecting epistatic interactions in genome-wide association studies, DOI: 10.1186 / 1471-2164-12-S2-S9.

Huang, K. & Nogueira, R., 2018. EpiRL: A Reinforcement Learning Agent to Facilitate Epistasis Detection, DOI: 10.1007/978-3-030-24409-5_19.

Jing, P.-J. & Shen, H.-B., 2014. MACOED: a multi-objective ant colony optimization algorithm for SNP epistasis detection in genome-wide association studies, DOI: 10.1093/bioinformatics/btu702.

Khronos, s.f. *OpenCL*. [En línea]
Available at: https://www.khronos.org/opencl/

Kim, Y., Choi, H. & Oh, H.-S., 2007. Smoothly clipped absolute deviation (SCAD), DOI: 10.1198/016214508000001066.

Kukreja, S. L., Löfberg, J. & Brenner, M. J., 2016. Least absolute shrinkage and selection operator (LASSO), DOI: 10.3182/20060329-3-AU-2901.00128.

Li, X., 2017. A fast and exhaustive method for heterogeneity and epistasis analysis based on multi-objective optimization, DOI: 10.1093/bioinformatics/btx339. 33(18).

Li, X., Liu, L., Zhou, J. & Wang, C., 2018. Heterogeneity Analysis and Diagnosis of Complex Diseases Based on Deep Learning Method, DOI: 10.1038/s41598-018-24588-5. 8(6155).

MathWorks, T., s.f. *Matlab.* [En línea]
Available at: https://www.mathworks.com/products/matlab.html

Moore, J. H. & Andrews, P. C., 2014. Epistasis Analysis Using Multifactor Dimensionality Reduction, DOI: 10.1007/978-1-4939-2155-3_16. En: *Methods in Molecular Biology.* s.l.:Springer.

Nickle, T. & Barrette-Ng, I., 2019. *Online Open Genetics (Nickle & Barrette-Ng).* [En línea]
Available at: https://bio.libretexts.org/Bookshelves/Genetics/Book:_Online_Open_Genetics_(Nickle_and_Barrette-Ng)

OpenMP, s.f. *OpenMP.* [En línea]
Available at: https://www.openmp.org/

Piette, E. R. & Moore, J. H., 2018. Improving machine learning reproducibility in genetic association studies with proportional instance cross validation (PICV), DOI: 10.1186/s13040-018-0167-7. 11(6).

Purcell, S. y otros, 2007. PLINK: A Tool Set for Whole-Genome Association and Population-Based Linkage Analyses, DOI: 10.1086/519795.

Tuo, S. y otros, 2016. FHSA-SED: Two-Locus Model Detection for Genome-Wide Association Study with Harmony Search Algorithm, DOI: 10.1371/journal.pone.0150669. 11(3).

Wan, X. y otros, 2010. BOOST: A Fast Approach to Detecting Gene-Gene Interactions in Genome-wide Case-Control Studies, DOI: 10.1016/j.ajhg.2010.07.021. 87(3).

Yang, C.-H., Chuang, L.-Y. & Lin, Y.-D., 2018. Multiobjective multifactor dimensionality reduction to detect SNP–SNP interactions, DOI: 10.1093/bioinformatics/bty076. 34(13).

ANEXO

Algoritmos Importantes de la Versión Final

Programa principal (.cpp):

LeerMatrizVectorDeArchivo:

```
1   //Lee los datos a partir de un archivo y los guarda en un vector de forma comprimida
2   int leerMatrizVectorDeArchivo(string fileName, int nCol, char *matriz, char *ultimaCol)//800*101
3   {
4       int i = 0;//indice para guardar en la matriz
5       int j = 0;//indice para guardar en el vector estado(ultima columna)
6       int bucle;
7       int cont = 0;
8       int salida = 0;
9       char data;
10      char aux = 0;
11
12      ifstream inputFile(fileName.c_str());
13      if (inputFile.is_open())
14      {
15          while (!inputFile.eof())
16          {
17              bucle = 0;
18              while(bucle < 4)
19              {
20                  if(!inputFile.eof())
21                  {
22                      inputFile.get(data);
23                      //cout<<data;
24                      if (data != '\n' && data != ',' && data != 13)
25                      {
26                          if (cont == nCol - 1)
27                          {
28                              ultimaCol[j] = data - 48;
29                              j++;
30                              cont = 0;
31                          }
32                          else
33                              cont++;
34
35                          data = data - 48;
36                          aux += data << (2 * bucle);
37                          bucle++;
38                      }
39                  }
```

```
40                  else
41                  {
42                      bucle++;
43                  }
44              }
45              matriz[i] = aux;
46              aux = 0;
47              i++;
48
49          }
50          cout << endl;
51          inputFile.close();
52      }
53      else
54      {
55          cout << "Fallo al abrir el archivo" << endl;
56          salida = 1;
57      }
58
59      return salida;
60  }
```

Exhaustive_search2:

```
1   int OpenCL_exhausive_search2(char devTipe, char *matriz, char *estado, int dim_epi, int nFil, int nCol, int nColComp, float* My_Factorial, int &index_FilterSnps, int *bestsolutionsPos, float *bestsolutionsScore,
2           int workitems, int numCombinaciones, int &contVentana_total, int &shared_i, int &shared_j, int &shared_k, int platform_used_id, int device_used_id, string device_name) {
3
4       //Inicializamos las variables y arrays.s
5       duration<double, std::milli> time_span;
6       int n_in = nCol - 1;
7       int tam_snp_pos_in = dim_epi;
8       int indice_combinacion = 0;
9       int contVentana = 0;
10      int contComb = 0;
11
12      float *all_score = new float[(numCombinaciones + 4) * 2];//tamano ventana * columnas + los 4 últimos para guardar los resultados de la ventana anterior
13      int *all_pos = new int[(numCombinaciones + 4) * dim_epi];//tamano ventana * columnas + los 4 últimos para guardar los resultados de la ventana anterior
14      //int *snp_pos = new int[tam_snp_pos_in * (n_in - 2)];
15      char *snp_com = new char[nFil * tam_snp_pos_in * (n_in - 2)];
16      char *mR_concatenarMatrices = new char[(1 + tam_snp_pos_in) * nFil * (n_in - 2)];
17      char *matrizConcatenadaSinEstado = new char[(nFil*tam_snp_pos_in) * (n_in - 2)];
18      //int *bestsolutionsPos = new int[4 * 3];
19      //float *bestsolutionsScore = new float[4 * 2];
20      int *vectorCombinaciones = new int[numCombinaciones*dim_epi];
21
22      //Inicializamos las variables para OpenCL.
23      cl_platform_id *platform_ids;
24      cl_device_id *device_ids;
25      cl_context context;
26      cl_command_queue command_queue;
27      cl_program program;
28      cl_kernel kernel;
29
30      cl_uint ret_num_platforms;
```

```
31      cl_uint ret_num_devices;
32
33      FILE *fp;
34      char *source_str;
35      size_t source_size;
36
37      cl_mem matriz_in;
38      cl_mem estado_in;
39      cl_mem my_factorial_in;
40      cl_mem vector_de_unos_in;
41      cl_mem all_score_in;
42      cl_mem all_pos_in;
43      //cl_mem snp_pos_in;
44      cl_mem snp_com_in;
45      cl_mem mR_concatenarMatrices_in;
46      cl_mem matrizConcatenadaSinEstado_in;
47      cl_mem vectorCombinaciones_in;
48
49      cl_ulong time_start;
50      cl_ulong time_end;
51      double nanoSeconds;
52
53
54      llenarVectorFloat((numCombinaciones + 4) * 2, all_score, 100000);//se llena con 100000 en
    cada elemento
55      llenarVector((numCombinaciones + 4)*dim_epi, all_pos, 0);//se llena con ceros en cada
    elemento
56      llenarVector(4 * dim_epi, bestsolutionsPos, 0);
57      llenarVectorFloat(4 * 2, bestsolutionsScore, 100000);
58      llenarVector(numCombinaciones*dim_epi, vectorCombinaciones, -1);
59
60
61      ////////////////////////////////////////////////////////////////////////////////////////
62      ////////////////////////////  Comienzo OpenCL  /////////////////////////////////////////////
63      ////////////////////////////////////////////////////////////////////////////////////////
64
65      //Obtenemos el numero de plataformas.
66      cl_int ret = clGetPlatformIDs(0, NULL, &ret_num_platforms);
67      if (!comprobarError(ret))
68      {
69          cout << "Error al obtener el numero de plataformas" << endl;
70          return 1;
71      }
72
73      //Mostramos el numero de plataformas.
74      //cout << "Numero de plataformas: " << ret_num_platforms << endl;
75
76      //Reservamos Memoria para el array de plataformas.
77      platform_ids = (cl_platform_id*)malloc(sizeof(cl_platform_id) * ret_num_platforms);
78
79      //Obtenemos las plataformas en el array platform_ids.
80      ret = clGetPlatformIDs(ret_num_platforms, platform_ids, NULL);
81      if (!comprobarError(ret))
82      {
83          cout << "Error al obtener las plataformas" << endl;
84          return 1;
85      }
86
87      //Obtenemos información sobre las plataformas.
88      //ret = obtenerInfoPlataformas(platform_ids, ret_num_platforms);
```

```
89
90      //Obtenemos el numero de dispositivos.
91      ret = clGetDeviceIDs(platform_ids[platform_used_id], CL_DEVICE_TYPE_ALL, 0, NULL,
    &ret_num_devices);
92      if (!comprobarError(ret))
93      {
94          cout << "Error al obtener el numero de dispositivos" << endl;
95          return 1;
96      }
97
98      //Mostramos el numero de dispositivos GPU.
99      //cout << "Numero de dispositivos GPU: " << ret_num_devices << endl;
100
101         //Reservamos Memoria para el array de plataformas.
102         device_ids = (cl_device_id*)malloc(sizeof(cl_device_id) * ret_num_devices);
103
104         //Obtenemos los dispositivos GPU en el array device_ids.
105         ret = clGetDeviceIDs(platform_ids[platform_used_id], CL_DEVICE_TYPE_ALL,
    ret_num_devices, device_ids, NULL);
106         if (!comprobarError(ret))
107         {
108             cout << "Error al obtener el numero de dispositivos" << endl;
109             return 1;
110         }
111
112         //Obtenemos información sobre los dispositivos.
113         //ret = obtenerInfoDispositivos(device_ids, ret_num_devices);
114
115
116         //Creamos el contexto.
117         context = clCreateContext(NULL, ret_num_devices, device_ids, NULL, NULL, &ret);
118         if (!comprobarError(ret))
119         {
120             cout << "Error al crear el contexto" << endl;
121             return 1;
122         }
123
124         //Cremos las colas de comandos.
125         /*
126         for (int i = 0; i < ret_num_devices; i++)
127         {
128             command_queue = clCreateCommandQueue(context, device_ids[i], 0, &ret);
129         }
130         */
131
132         command_queue = clCreateCommandQueue(context, device_ids[device_used_id],
    CL_QUEUE_PROFILING_ENABLE, &ret);
133
134         //Cargamos el codigo fuente del kernel en el array source_str.
135         fp = fopen("programa.cl", "r");
136         if (!fp) {
137             fprintf(stderr, "Fallo al cargar el kernel.\n");
138             exit(1);
139         }
140         source_str = (char*)malloc(MAX_SOURCE_SIZE);
141         source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
142         fclose(fp);
143
144         //Creamos el objeto programa.
145         program = clCreateProgramWithSource(context, 1,
```

```
146             (const char **)&source_str, (const size_t *)&source_size, &ret);
147         if (!comprobarError(ret))
148         {
149             cout << "Error al crear el programa." << endl;
150             return 1;
151         }
152 
153         //Compilamos el objeto programa.
154         ret = clBuildProgram(program, ret_num_devices, device_ids, NULL, NULL, NULL);
155         if (!comprobarError(ret))
156         {
157             cout << "Error al compilar programa." << endl;
158             ret = obtenerInfoCompilacion(device_ids, ret_num_devices, program);
159             return 1;
160         }
161 
162         //Capturamos los errores de compilacion.
163         //ret = obtenerInfoCompilacion(device_ids, ret_num_devices, program);
164 
165         //Creamos el objeto kernel, utilizamos el primer parametro para diferenciar entre el kernel del CPU y el de GPU.
166         if (devTipe == 'G')
167         {
168             kernel = clCreateKernel(program, "exhausive_search_GPU", &ret);
169             if (!comprobarError(ret))
170             {
171                 cout << "Error al crear el objeto kernel." << endl;
172                 return 1;
173             }
174         }
175         else if (devTipe == 'C')
176         {
177             kernel = clCreateKernel(program, "exhausive_search_CPU", &ret);
178             if (!comprobarError(ret))
179             {
180                 cout << "Error al crear el objeto kernel." << endl;
181                 return 1;
182             }
183         }
184         else
185             cout << "Error al crear el objeto kernel, no se selecciono ni CPU ni GPU." << endl;
186         //Definimos el numero de WorkItems.
187         size_t global_item_size = workitems; // Numero total de work_items
188                                                 //size_t local_item_size = n_in - 2;//n_in - 2; // Numero de work_items por grupo
189 
190         //Creamos los objetos de memoria.
191         matriz_in = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(char)*(nFil * nColComp), NULL, &ret); /////COMPRIMIDA
192         if (!comprobarError(ret))
193         {
194             cout << "Error al crear el buffer para matriz." << endl;
195             return 1;
196         }
197 
198         estado_in = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(char)*nFil, NULL, &ret);
199         if (!comprobarError(ret))
200         {
201             cout << "Error al crear el buffer para estado." << endl;
```

```
202             return 1;
203         }
204
205         my_factorial_in = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float)*nFil,
    NULL, &ret);
206         if (!comprobarError(ret))
207         {
208             cout << "Error al crear el buffer para My_Factorial." << endl;
209             return 1;
210         }
211
212         all_score_in = clCreateBuffer(context, CL_MEM_READ_WRITE,
    sizeof(float)*((numCombinaciones + 4) * 2), NULL, &ret);
213         if (!comprobarError(ret))
214         {
215             cout << "Error al crear el buffer para all_score." << endl;
216             return 1;
217         }
218
219         all_pos_in = clCreateBuffer(context, CL_MEM_READ_WRITE,
    sizeof(int)*((numCombinaciones + 4)*dim_epi), NULL, &ret);
220         if (!comprobarError(ret))
221         {
222             cout << "Error al crear el buffer para all_pos." << endl;
223             return 1;
224         }
225
226         snp_com_in = clCreateBuffer(context, CL_MEM_READ_WRITE,
    sizeof(char)*(tam_snp_pos_in*nFil)*global_item_size, NULL, &ret);
227         if (!comprobarError(ret))
228         {
229             cout << "Error al crear el buffer para snp_com." << endl;
230             return 1;
231         }
232
233         mR_concatenarMatrices_in = clCreateBuffer(context, CL_MEM_READ_WRITE,
    sizeof(char)*((1 + tam_snp_pos_in) * nFil)*global_item_size, NULL, &ret);
234         if (!comprobarError(ret))
235         {
236             cout << "Error al crear el buffer para mR_concatenarMatrices." << endl;
237             return 1;
238         }
239
240         matrizConcatenadaSinEstado_in = clCreateBuffer(context, CL_MEM_READ_WRITE,
    sizeof(char)*(tam_snp_pos_in * nFil)*global_item_size, NULL, &ret);
241         if (!comprobarError(ret))
242         {
243             cout << "Error al crear el buffer para matrizConcatenadaSinEstado." << endl;
244             return 1;
245         }
246
247         vectorCombinaciones_in = clCreateBuffer(context, CL_MEM_READ_WRITE,
    sizeof(int)*(numCombinaciones*dim_epi), NULL, &ret);
248         if (!comprobarError(ret))
249         {
250             cout << "Error al crear el buffer para vectorCombinaciones_in." << endl;
251             return 1;
252         }
253
254         //Transferimos los datos al dispositivo a traves de los buffers.
```

```
255
256         ret = clEnqueueWriteBuffer(command_queue, matriz_in, CL_TRUE, 0, sizeof(char)*(nFil *
    nColComp),
257             matriz, 0, NULL, NULL);
258         if (!comprobarError(ret))
259         {
260             cout << "Error al transferir matriz a Buffer matriz_in." << endl;
261             return 1;
262         }
263
264         ret = clEnqueueWriteBuffer(command_queue, estado_in, CL_TRUE, 0, sizeof(char)*nFil,
265             estado, 0, NULL, NULL);
266         if (!comprobarError(ret))
267         {
268             cout << "Error al transferir estado a Buffer estado_in." << endl;
269             return 1;
270         }
271
272         ret = clEnqueueWriteBuffer(command_queue, my_factorial_in, CL_TRUE, 0,
    sizeof(float)*nFil,
273             My_Factorial, 0, NULL, NULL);
274         if (!comprobarError(ret))
275         {
276             cout << "Error al transferir My_Factorial a Buffer my_factorial_in." << endl;
277             return 1;
278         }
279
280         ret = clEnqueueWriteBuffer(command_queue, all_score_in, CL_TRUE, 0,
    sizeof(float)*((numCombinaciones + 4) * 2),
281             all_score, 0, NULL, NULL);
282         if (!comprobarError(ret))
283         {
284             cout << "Error al transferir all_score a Buffer all_score_in." << endl;
285             return 1;
286         }
287
288         ret = clEnqueueWriteBuffer(command_queue, all_pos_in, CL_TRUE, 0,
    sizeof(int)*((numCombinaciones + 4)*dim_epi),
289             all_pos, 0, NULL, NULL);
290         if (!comprobarError(ret))
291         {
292             cout << "Error al transferir all_pos a Buffer all_pos_in." << endl;
293             return 1;
294         }
295
296         //Enlazamos los objetos de memoria con los parámetros del kernel.
297         ret = clSetKernelArg(kernel, 0, sizeof(int), &indice_combinacion);
298         if (!comprobarError(ret))
299         {
300             cout << "Error al enlazar nglobal_item_size_in entrada al parametro del kernel." << endl;
301             return 1;
302         }
303
304         ret = clSetKernelArg(kernel, 1, sizeof(int), &n_in);
305         if (!comprobarError(ret))
306         {
307             cout << "Error al enlazar n_in entrada al parametro del kernel." << endl;
308             return 1;
309         }
310
```

```
311     ret = clSetKernelArg(kernel, 2, sizeof(int), &tam_snp_pos_in);
312     if (!comprobarError(ret))
313     {
314         cout << "Error al enlazar tam_snp_pos_in al parametro del kernel." << endl;
315         return 1;
316     }
317
318     ret = clSetKernelArg(kernel, 3, sizeof(int), &nFil);
319     if (!comprobarError(ret))
320     {
321         cout << "Error al enlazar nFil al parametro del kernel." << endl;
322         return 1;
323     }
324
325     ret = clSetKernelArg(kernel, 4, sizeof(int), &nCol);
326     if (!comprobarError(ret))
327     {
328         cout << "Error al enlazar nCol al parametro del kernel." << endl;
329         return 1;
330     }
331
332     ret = clSetKernelArg(kernel, 5, sizeof(cl_mem), (void *)&matriz_in);
333     if (!comprobarError(ret))
334     {
335         cout << "Error al enlazar el buffer matriz_in al parametro del kernel." << endl;
336         return 1;
337     }
338
339     ret = clSetKernelArg(kernel, 6, sizeof(cl_mem), (void *)&estado_in);
340     if (!comprobarError(ret))
341     {
342         cout << "Error al enlazar el buffer estado_in al parametro del kernel." << endl;
343         return 1;
344     }
345
346     ret = clSetKernelArg(kernel, 7, sizeof(cl_mem), (void *)&my_factorial_in);
347     if (!comprobarError(ret))
348     {
349         cout << "Error al enlazar el buffer my_factorial_in al parametro del kernel." << endl;
350         return 1;
351     }
352
353     ret = clSetKernelArg(kernel, 8, sizeof(cl_mem), (void *)&all_score_in);
354     if (!comprobarError(ret))
355     {
356         cout << "Error al enlazar el buffer all_score_in al parametro del kernel." << endl;
357         return 1;
358     }
359
360     ret = clSetKernelArg(kernel, 9, sizeof(cl_mem), (void *)&all_pos_in);
361     if (!comprobarError(ret))
362     {
363         cout << "Error al enlazar el buffer all_pos_in al parametro del kernel." << endl;
364         return 1;
365     }
366
367     ret = clSetKernelArg(kernel, 10, sizeof(cl_mem), (void *)&snp_com_in);
368     if (!comprobarError(ret))
369     {
370         cout << "Error al enlazar el buffer snp_com_in al parametro del kernel." << endl;
```

```
371             return 1;
372         }
373
374         ret = clSetKernelArg(kernel, 11, sizeof(cl_mem), (void *)&mR_concatenarMatrices_in);
375         if (!comprobarError(ret))
376         {
377             cout << "Error al enlazar el buffer mR_concatenarMatrices_in al parametro del kernel." << endl;
378             return 1;
379         }
380
381         ret = clSetKernelArg(kernel, 12, sizeof(cl_mem), (void *)&matrizConcatenadaSinEstado_in);
382         if (!comprobarError(ret))
383         {
384             cout << "Error al enlazar el buffer matrizConcatenadaSinEstado_in al parametro del kernel." << endl;
385             return 1;
386         }
387
388         ret = clSetKernelArg(kernel, 13, sizeof(int), &numCombinaciones);
389         if (!comprobarError(ret))
390         {
391             cout << "Error al enlazar numCombinacionesal parametro del kernel." << endl;
392             return 1;
393         }
394
395
396         //////////////////////  LLamamos al Kernel con las ventanas  ///////////////////
397
398         high_resolution_clock::time_point t1 = high_resolution_clock::now();
399         cl_event event;
400
401         while (shared_i < n_in - 2)
402         {
403
404                 #pragma omp critical //Como se van a hacer varias cosas, tiene que usarse critical
405                 {
406                     if(devTipe == 'C')
407                     {
408                         vectorCombinaciones[contComb*dim_epi] = shared_i;
409                         vectorCombinaciones[(contComb*dim_epi) + 1] = shared_j;
410                         vectorCombinaciones[(contComb*dim_epi) + 2] = shared_k;
411                     }
412                     else if (devTipe == 'G')
413                     {
414                         vectorCombinaciones[contComb] = shared_i;
415                         vectorCombinaciones[contComb+numCombinaciones] = shared_j;
416                         vectorCombinaciones[contComb + 2*numCombinaciones] = shared_k;
417                     }
418                     else
419                         cout << "Error al almacenar combinaciones"<<endl;
420
421                     if (shared_k + 1 < n_in)
422                     {
423                         shared_k++;
424                     }
425                     else
426                     {
427                         if (shared_j + 1 < n_in - 1)
```

```
428                             {
429                                 shared_j++;
430                             }
431                             else
432                             {
433                                 shared_i++;
434                                 shared_j = shared_i + 1;
435 
436                             }
437                             shared_k = shared_j + 1;
438                         }
439                     }
440 
441                     contComb++;
442 
443                     if (contComb == numCombinaciones)
444                     {
445                         contVentana++;
446 
447                         #pragma omp atomic
448                         contVentana_total++;
449 
450                         //Definimos los objetos de memoria de la ventana: el vector combinaciones y el contador:
451 
452                         ret = clEnqueueWriteBuffer(command_queue, vectorCombinaciones_in, CL_TRUE, 0, sizeof(int)*(numCombinaciones*dim_epi),
453                             vectorCombinaciones, 0, NULL, NULL);
454                         if (!comprobarError(ret))
455                         {
456                             cout << "Error al transferir vectorCombinaciones a Buffer vectorCombinaciones_in." << endl;
457                             return 1;
458                         }
459 
460                         ret = clSetKernelArg(kernel, 14, sizeof(cl_mem), (void *)&vectorCombinaciones_in);
461                         if (!comprobarError(ret))
462                         {
463                             cout << "Error al enlazar el buffer vectorCombinaciones_in al parametro del kernel." << endl;
464                             return 1;
465                         }
466 
467                         ret = clSetKernelArg(kernel, 15, sizeof(int), &contComb);
468                         if (!comprobarError(ret))
469                         {
470                             cout << "Error al enlazar contComb al parametro del kernel." << endl;
471                             return 1;
472                         }
473 
474                         ////////////////////AllPos y AllScore
475                         ret = clEnqueueWriteBuffer(command_queue, all_score_in, CL_TRUE, 0, sizeof(float)*((numCombinaciones + 4) * 2),
476                             all_score, 0, NULL, NULL);
477                         if (!comprobarError(ret))
478                         {
479                             cout << "Error al transferir all_score a Buffer all_score_in." << endl;
480                             return 1;
481                         }
```

```
482
483                     ret = clEnqueueWriteBuffer(command_queue, all_pos_in, CL_TRUE, 0, sizeof(int)*((numCombinaciones + 4)*dim_epi),
484                         all_pos, 0, NULL, NULL);
485                     if (!comprobarError(ret))
486                     {
487                         cout << "Error al transferir all_pos a Buffer all_pos_in." << endl;
488                         return 1;
489                     }
490
491                     ret = clSetKernelArg(kernel, 0, sizeof(int), &indice_combinacion);
492                     if (!comprobarError(ret))
493                     {
494                         cout << "Error al enlazar nglobal_item_size_in entrada al parametro del kernel." << endl;
495                         return 1;
496                     }
497
498                     //Ejecutamos el kernel OpenCL utilizando una dimensión
499                     ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
500                         &global_item_size, NULL, 0, NULL, &event);
501                     if (!comprobarError(ret))
502                     {
503                         cout << "Error al ejecutar el kernel." << endl;
504                         return 1;
505                     }
506
507                     //Esperamos a que finalicen todos los work items
508                     ret = clFinish(command_queue);
509                     if (!comprobarError(ret))
510                     {
511                         cout << "Error al finalizar." << endl;
512                         return 1;
513                     }
514
515                     //Leemos los buffer de salida.
516                     ret = clEnqueueReadBuffer(command_queue, all_score_in, CL_TRUE, 0,
517                         sizeof(float)*((numCombinaciones + 4) * 2), all_score, 0, NULL, NULL);
518                     if (!comprobarError(ret))
519                     {
520                         cout << "Error al leer el buffer de salida de all_score." << endl;
521                         return 1;
522                     }
523
524                     ret = clEnqueueReadBuffer(command_queue, all_pos_in, CL_TRUE, 0,
525                         sizeof(int)*((numCombinaciones + 4)*dim_epi), all_pos, 0, NULL, NULL);
526                     if (!comprobarError(ret))
527                     {
528                         cout << "Error al leer el buffer de salida de all_pos." << endl;
529                         return 1;
530                     }
531
532                     //Obtenemos bestsolutions.
533                     index_FilterSnps = non_dominant_top_k(all_score, numCombinaciones + 4, 2, all_pos, numCombinaciones + 4, dim_epi, bestsolutionsPos, bestsolutionsScore); //aS,100,2,aP,100,3 //n_in sera numWI * elementos*WI
534
535                     //Reinicializamos allScore y allPos:
```

```
536                     llenarVectorFloat((numCombinaciones + 4) * 2, all_score, 100000);//se llena con 100000 en cada elemento
537                     llenarVector((numCombinaciones + 4)*dim_epi, all_pos, 0);//se llena con ceros en cada elemento
538
539
                    //numVentana++;
540                     indice_combinacion = indice_combinacion + contComb;
541
542                     contComb = 0;
543                     /*
544                     //mostrar cada 100 ventanas
545                     if (contVentana % 100 == 0)
546                         cout << contVentana << " " <<endl;
547                     //printf("%d \n",contVentana);
548                     */
549
550                   } //Fin if
551
552
553         }
554
555         if (contComb != 0)
556         {
557             contVentana++;
558
559             #pragma omp atomic
560             contVentana_total++;
561
562             //Ejecutamos el kernel  una vez más con el resto de combinaciones
563
564             //Definimos los objetos de memoria de la ventana: el vector combinaciones y el contador:
565
566             ret = clEnqueueWriteBuffer(command_queue, vectorCombinaciones_in, CL_TRUE, 0, sizeof(int)*(numCombinaciones*dim_epi),
567                 vectorCombinaciones, 0, NULL, NULL);
568             if (!comprobarError(ret))
569             {
570                 cout << "Error al transferir vectorCombinaciones a Buffer vectorCombinaciones_in." << endl;
571                 return 1;
572             }
573
574             ret = clSetKernelArg(kernel, 14, sizeof(cl_mem), (void *)&vectorCombinaciones_in);
575             if (!comprobarError(ret))
576             {
577                 cout << "Error al enlazar el buffer vectorCombinaciones_in al parametro del kernel." << endl;
578                 return 1;
579             }
580
581             ret = clSetKernelArg(kernel, 15, sizeof(int), &contComb);
582             if (!comprobarError(ret))
583             {
584                 cout << "Error al enlazar contComb al parametro del kernel." << endl;
585                 return 1;
586             }
587
588             ////////////////////AllPos y AllScore
```

```
589             ret = clEnqueueWriteBuffer(command_queue, all_score_in, CL_TRUE, 0,
    sizeof(float)*((numCombinaciones + 4) * 2),
590                 all_score, 0, NULL, NULL);
591             if (!comprobarError(ret))
592             {
593                 cout << "Error al transferir all_score a Buffer all_score_in." << endl;
594                 return 1;
595             }
596
597             ret = clEnqueueWriteBuffer(command_queue, all_pos_in, CL_TRUE, 0,
    sizeof(int)*((numCombinaciones + 4)*dim_epi),
598                 all_pos, 0, NULL, NULL);
599             if (!comprobarError(ret))
600             {
601                 cout << "Error al transferir all_pos a Buffer all_pos_in." << endl;
602                 return 1;
603             }
604
605             ret = clSetKernelArg(kernel, 0, sizeof(int), &indice_combinacion);
606             if (!comprobarError(ret))
607             {
608                 cout << "Error al enlazar nglobal_item_size_in entrada al parametro del kernel." <<
    endl;
609                 return 1;
610             }
611
612             //Ejecutamos el kernel OpenCL utilizando una dimensión
613             ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
614                 &global_item_size, NULL, 0, NULL, &event);
615             if (!comprobarError(ret))
616             {
617                 cout << "Error al ejecutar el kernel." << endl;
618                 return 1;
619             }
620
621             //Esperamos a que finalicen todos los work items
622             clWaitForEvents(1, &event);
623
624             ret = clFinish(command_queue);
625             if (!comprobarError(ret))
626             {
627                 cout << "Error al finalizar." << endl;
628                 return 1;
629             }
630
631             //Obtenemnos el tiemnpo de ejecución de kernel
632             clGetEventProfilingInfo(event,                CL_PROFILING_COMMAND_START,
    sizeof(time_start), &time_start, NULL);
633             clGetEventProfilingInfo(event,                  CL_PROFILING_COMMAND_END,
    sizeof(time_end), &time_end, NULL);
634
635             nanoSeconds = time_end-time_start;
636
637             //Leemos los buffer de salida.
638             ret = clEnqueueReadBuffer(command_queue, all_score_in, CL_TRUE, 0,
639                 sizeof(float)*((numCombinaciones + 4) * 2), all_score, 0, NULL, NULL);
640             if (!comprobarError(ret))
641             {
642                 cout << "Error al leer el buffer de salida de all_score." << endl;
643                 return 1;
```

```
644             }
645
646             ret = clEnqueueReadBuffer(command_queue, all_pos_in, CL_TRUE, 0,
647                 sizeof(int)*((numCombinaciones + 4)*dim_epi), all_pos, 0, NULL, NULL);
648             if (!comprobarError(ret))
649             {
650                 cout << "Error al leer el buffer de salida de all_pos." << endl;
651                 return 1;
652             }
653
654             index_FilterSnps = non_dominant_top_k(all_score, numCombinaciones + 4, 2, all_pos,
    numCombinaciones + 4, dim_epi, bestsolutionsPos, bestsolutionsScore); //aS,100,2,aP,100,3 //n_in
    sera numWI * elementos*WI
655         }
656         high_resolution_clock::time_point t2 = high_resolution_clock::now();
657         time_span = t2 - t1;
658
659         cout<<"Solucion parcial de dispositivo "<< device_name <<endl;
660
661         cout << "Solucion Pos:" << endl;
662         mostrarMatrizDeVector(bestsolutionsPos, index_FilterSnps, 3);
663         cout << endl << endl;
664
665         cout << "Solucion Score:" << endl;
666         mostrarMatrizDeVectorFloat(bestsolutionsScore, index_FilterSnps, 2);
667         cout << endl;
668
669         //std::cout << "Tiempo de ejecución del kernel en milisegundos" << nanoSeconds/1000000
    <<endl;
670
671         std::cout << "Tiempo de ejecucion del dispositivo " << device_name << " en milisegundos:
    " << time_span.count() <<endl;
672
673         cout << "Numero de ventanas ejecutadas: " << contVentana << endl;
674         cout << endl;
675
676         //Liberamos la memoria:
677         delete(all_score);
678         delete(all_pos);
679         //delete(snp_pos);
680         delete(snp_com);
681         delete(mR_concatenarMatrices);
682         delete(matrizConcatenadaSinEstado);
683         delete(bestsolutionsPos);
684         delete(bestsolutionsScore);
685         delete(vectorCombinaciones);
686
687         //Liberacion de OpenCl:
688         ret = clFlush(command_queue);
689         ret = clReleaseKernel(kernel);
690         ret = clReleaseProgram(program);
691         ret = clReleaseMemObject(matriz_in);
692         ret = clReleaseMemObject(estado_in);
693         ret = clReleaseMemObject(my_factorial_in);
694         ret = clReleaseMemObject(all_score_in);
695         ret = clReleaseMemObject(all_pos_in);
696         //ret = clReleaseMemObject(snp_pos_in);
697         ret = clReleaseMemObject(snp_com_in);
698         ret = clReleaseMemObject(mR_concatenarMatrices_in);
699         ret = clReleaseMemObject(matrizConcatenadaSinEstado_in);
```

```
700         ret = clReleaseMemObject(vectorCombinaciones_in);
701         ret = clReleaseCommandQueue(command_queue);
702         ret = clReleaseContext(context);
703
704         free(device_ids);
705         free(platform_ids);
706
707
708
709         ////////////////////////////////////////////////////////////////////////////////////
710         ////////////////////////////////  Fin OpenCL  ////////////////////////////////////////////////
711         ////////////////////////////////////////////////////////////////////////////////////
712
713
714         return 0;//matriz de 3x3
715     }
```

Main:

```
1   int main(int argc, char *argv[])
2   {
3       //Comprobamos que los argumentos del programa son correctos.
4       if (argc != 5)
5       {
6           cout << "Argumentos incorrectos:" << endl;
7           cout << "Argumentos necesarios: Ruta de db, Numero de filas (Pacientes), Numero de
    Columnas (SNPs + 1[la columna del estado], Numero de Combinaciones por ventana" << endl;
8           return 1;
9       }
10
11      //Inicializamos las variables.
12      int samplesize = 1600; //tamano muestra
13      float Bp = 0.1; //
14      int dim_epi = 3; //3
15      int dim = 100; //100
16      string numFilString = argv[2];
17      int numFil = std::atoi(numFilString.c_str());
18      string numColString = argv[3];
19      int numCol = std::atoi(numColString.c_str());
20      string numCombxVentanaString = argv[4];
21      int numCombxVentana = std::atoi(numCombxVentanaString.c_str());
22      int shared_i = 0;
23      int shared_j = shared_i + 1;
24      int shared_k = shared_j + 1;
25
26      int numCol_comp;
27
28      if (numCol % 4 != 0)
29      {
30          numCol_comp = numCol / 4 + 1;
31      }
32      else
33          numCol_comp = numCol / 4;
34
35      int comb = binomialCoeff(dim, dim_epi);
36      float pvalue = Bp / comb;
37
38      float *vFactorial = new float[numFil];
39      char *estado = new char[numFil];//vector con la ultima columna de la matriz
```

```
40      char *matriz = new char[numFil*numCol_comp];//800*101, comprimidas = 800* colComp(col/4 si col%4 ==0 o col/4 + 1 si no)
41
42      int ac = 0;
43      int ac_inter = 0;
44      int ac_inter2 = 0;
45      int ac_inter3 = 0;
46      int functionSNP[] = { 97, 98, 99 }; //H1
47      int functionSNP2[] = { 94, 95, 96 }; //H2
48      int functionSNP3[] = { 91, 92, 93 }; //H3
49      int index_FilterSnps;
50
51      int *bestsolutionsPosDevice_1 = new int[4 * 3];
52      float *bestsolutionsScoreDevice_1 = new float[4 * 2];
53
54      int *bestsolutionsPosDevice_2 = new int[4 * 3];
55      float *bestsolutionsScoreDevice_2 = new float[4 * 2];
56
57      int *bestsolutionsPosDevice_3 = new int[4 * 3];
58      float *bestsolutionsScoreDevice_3 = new float[4 * 2];
59
60      int contVentana_total = 0;
61
62      duration<double, std::milli> time_span_total;
63      high_resolution_clock::time_point t1_total = high_resolution_clock::now();
64
65      My_factorial(numFil, vFactorial);
66      if (leerMatrizVectorDeArchivo(argv[1], numCol, matriz, estado) == 1)
67      {
68          cout << "Fallo al abrir el archivo!!!!!!!!!!!!!" << endl;
69          return 1;
70      }
71
72      else
73      {
74          cout << "Archivo leido correctamente" << endl;
75      }
76      printf("filas %d, cols %d\n\n", numFil, numCol);
77
78      #pragma omp parallel private(index_FilterSnps) num_threads(3)
79      {
80          #pragma omp sections
81          {
82              #pragma omp section
83              {
84                  if (OpenCL_exhausive_search2('G', matriz, estado, dim_epi, numFil, numCol, numCol_comp, vFactorial, index_FilterSnps, bestsolutionsPosDevice_1, bestsolutionsScoreDevice_1, device_1_WI,
85                          numCombxVentana, contVentana_total, shared_i, shared_j, shared_k, 0, 0, "GPU Titan X") != 0) //TitanX
86                  {
87                      cout << "Error al ejecutar OpenCL_exhausive_search2 con GPU TitanX";
88                  }
89              }
90
91              #pragma omp section
92              {
93                  if (OpenCL_exhausive_search2('G', matriz, estado, dim_epi, numFil, numCol, numCol_comp, vFactorial, index_FilterSnps, bestsolutionsPosDevice_2, bestsolutionsScoreDevice_2, device_2_WI,
```

```
94                          numCombxVentana, contVentana_total, shared_i, shared_j, shared_k, 0, 1, "GPU K40m") != 0) //K40m
95                  {
96                      cout << "Error al ejecutar OpenCL_exhausive_search2 con GPU K40m";
97                  }
98              }
99
100                 #pragma omp section
101                 {
102                     if (OpenCL_exhausive_search2('C', matriz, estado, dim_epi, numFil, numCol, numCol_comp, vFactorial, index_FilterSnps, bestsolutionsPosDevice_3, bestsolutionsScoreDevice_3, device_3_WI,
103                                 numCombxVentana, contVentana_total, shared_i, shared_j, shared_k, 1, 0, "CPU") != 0) //CPU
104                     {
105                         cout << "Error al ejecutar OpenCL_exhausive_search2 con CPU";
106                     }
107                 }
108
109             }
110         }
111
112         high_resolution_clock::time_point t2_total = high_resolution_clock::now();
113         time_span_total = t2_total - t1_total;
114
115         cout << endl;
116         cout << "Resultados Totales: "<<endl;
117         std::cout << "Tiempo de ejecucion total en milisegundos: " << time_span_total.count() <<endl;
118         cout << "Numero de ventanas totales ejecutadas: " << contVentana_total << endl;
119         cout << endl;
120
121         //Liberacion de memoria:
122         delete(vFactorial);
123         delete(estado);
124         delete(matriz);
125
126         return 0;
127     }
```

Non_DominantTop_K:

```
1   int non_domiant_top_k(float *allScore, int nF_allScore, int nC_allScore, int *allPos, int nF_allPos, int nC_allPos, int* bestsolutionsPos, float* bestsolutionsScore)
2   {
3       int k = 3;
4       int index = 0;
5       //bestsolutions = new int[k * 3];
6       //llenarVector((k+1)*3,bestsolutionsPos,0);
7       float Y[2];
8       int U[2];
9
10      //mostrarMatrizDeVectorFloat(bestsolutionsScore, 4, 2);
11      //cout << "Fin bestsolutionsScore:" << endl;
12
13      //pasamos bestsolutionsPos y bestsolutionsScore a las últimas 3 filas de allPos y allScore
14      volcarBestSolutions(allScore, nF_allScore, nC_allScore, allPos, nF_allPos, nC_allPos, bestsolutionsPos, bestsolutionsScore, k + 1);
15
```

```
16      //mostrarMatrizDeVectorFloat(allScore, nF_allScore, nC_allScore);
17      //mostrarMatrizDeVector(allPos, nF_allPos, nC_allPos);
18      //cout << "Fin AllScore:" << endl;
19      //cout << endl << endl;
20
21      llenarVector((k + 1) * 3, bestsolutionsPos, 0);
22      llenarVectorFloat((k + 1) * 2, bestsolutionsScore, 10000);
23
24      while (index < k)//por ser c++ se empieza en 0 y acaba en 2
25      {
26          minimoMatriz2D(allScore, nF_allScore, nC_allScore, Y, U);
27
28          /*
29          cout << "para index = :" << index << endl;
30          mostrarMatrizDeVector(U, 1, 2);
31          mostrarMatrizDeVectorFloat(Y, 1, 2);
32          */
33          if (U[0] == U[1])
34          {
35
36              //Copiamos en bestsolutions la fila correspondiente de allPos
37
38              bestsolutionsPos[(index * 3)] = allPos[(U[0] * 3)];
39              bestsolutionsPos[(index * 3) + 1] = allPos[(U[0] * 3) + 1];
40              bestsolutionsPos[(index * 3) + 2] = allPos[(U[0] * 3) + 2];
41
42              bestsolutionsScore[(index * 2)] = allScore[(U[0] * 2)];
43              bestsolutionsScore[(index * 2) + 1] = allScore[(U[0] * 2) + 1];
44
45              //Se resetean las filas correspondientes en allPos y allScore
46              allPos[(U[0] * 3)] = 0;
47              allPos[(U[0] * 3) + 1] = 0;
48              allPos[(U[0] * 3) + 2] = 0;
49
50              allScore[(U[0] * 2)] = 0;
51              allScore[(U[0] * 2) + 1] = 0;
52
53              index++;
54          }
55          else
56          {
57
58              bestsolutionsPos[(index * 3)] = allPos[(U[0] * 3)];
59              bestsolutionsPos[(index * 3) + 1] = allPos[(U[0] * 3) + 1];
60              bestsolutionsPos[(index * 3) + 2] = allPos[(U[0] * 3) + 2];
61
62              bestsolutionsScore[(index * 2)] = allScore[(U[0] * 2)];
63              bestsolutionsScore[(index * 2) + 1] = allScore[(U[0] * 2) + 1];
64
65              allPos[(U[0] * 3)] = 0;
66              allPos[(U[0] * 3) + 1] = 0;
67              allPos[(U[0] * 3) + 2] = 0;
68
69              allScore[(U[0] * 2)] = 0;
70              allScore[(U[0] * 2) + 1] = 0;
71
72              index++;
73
74
75              //Lo hameos tambien con U[1]
```

```
76              bestsolutionsPos[(index * 3)] = allPos[(U[1] * 3)];
77              bestsolutionsPos[(index * 3) + 1] = allPos[(U[1] * 3) + 1];
78              bestsolutionsPos[(index * 3) + 2] = allPos[(U[1] * 3) + 2];
79
80              bestsolutionsScore[(index * 2)] = allScore[(U[1] * 2)];
81              bestsolutionsScore[(index * 2) + 1] = allScore[(U[1] * 2) + 1];
82
83              allPos[(U[1] * 3)] = 0;
84              allPos[(U[1] * 3) + 1] = 0;
85              allPos[(U[1] * 3) + 2] = 0;
86
87              allScore[(U[1] * 2)] = 0;
88              allScore[(U[1] * 2) + 1] = 0;
89
90              index++;
91          }
92
93      }
94      return index;
95  }
```

VolcarBestSolutions:

```
1   int volcarBestSolutions(float *allScore, int nF_allScore, int nC_allScore, int *allPos, int nF_allPos,
    int nC_allPos, int *bestsolutionsPos,
2       float *bestsolutionsScore, int k)
3   {
4       for (int i = 0; i<k; i++)
5       {
6           allPos[(nF_allPos - k + i)*nC_allPos] = bestsolutionsPos[i * 3];
7           allPos[(nF_allPos - k + i)*nC_allPos + 1] = bestsolutionsPos[(i * 3) + 1];
8           allPos[(nF_allPos - k + i)*nC_allPos + 2] = bestsolutionsPos[(i * 3) + 2];
9
10
11          allScore[(nF_allScore - k + i)*nC_allScore] = bestsolutionsScore[i * 2];
12          allScore[(nF_allScore - k + i)*nC_allScore + 1] = bestsolutionsScore[(i * 2) + 1];
13      }
14
15      return 0;
16  }
```

Archivo de OpenCL con los kernels (.cl):

Multiscore_Obj2:

```
1   int multiscore_obj2(int tamSnp_pos, __global char *snp_com, int desplazamiento_snp_com, int
    nFilas, int nCols, __constant char *estado, __constant float *My_factorial,
2       float *Score1, float *Score2, __global char *matrizA, int desplazamiento_matrizA, __global char
    *matrizConcSinEstado, int desplazamiento_matrizConcSinEstado, int i_in, int j_in, int k_in)
3   {
4       //MatrizA es la matriz concatenada de snp_com con estado.
5       int sample[4*4*4];
6       int matrizDisease[4*4*4];
7       int control[4*4*4];
```

```
8       concatenarMatrices(estado, 1, snp_com, desplazamiento_snp_com, tamSnp_pos, nFilas, matrizA, desplazamiento_matrizA);
9
10
11
12      float z = 0;
13      float y = 0;
14      float r = 0;
15      int disease_index = 0;
16      int control_index = 0;
17      float Score_aux;
18
19      Score_aux = mutualInfo_improved(matrizA, desplazamiento_matrizA, nFilas, (nCols+1), sample, matrizConcSinEstado, desplazamiento_matrizConcSinEstado);
20      Score_aux = 1/Score_aux;
21      (*Score1) = Score_aux;
22
23      accumarray3_estado(snp_com, desplazamiento_snp_com, estado, nFilas, matrizDisease);
24      restarMatrizes3D(matrizDisease, sample, control, 4, 4, 4);
25
26      sample[3*16+3*4+3] = 0;
27      matrizDisease[3*16+3*4+3] = 0;
28      control[3*16+3*4+3] = 0;
29
30      for(int i=0; i<3; i++)
31      {
32          for(int j=0; j<3; j++)
33          {
34              for(int k=0; k<3; k++)
35              {
36
37                  y = (My_factorial[sample[(i*16+j*4+k)]]) * 1000;
38                  if(matrizDisease[i*16+j*4+k] == 0)
39                  {
40                      disease_index = 1;
41                  }
42                  else
43                  {
44                      disease_index = matrizDisease[i*16+j*4+k];
45                  }
46
47                  if(control[i*16+j*4+k] == 0)
48                  {
49                      control_index = 1;
50                  }
51                  else
52                  {
53                      control_index = control[i*16+j*4+k];
54                  }
55                  r = (My_factorial[disease_index-1] + My_factorial[control_index-1]) * 1000;
56                  z = z + (r - y);
57
58              }
59          }
60      }
61
62      if(z < 0)//abs(z);
63      {
64          z = (float) (z * -1);
65      }
```

```
66      (*Score2) = z;
67
68      //delete(matrizA);
69      return 0;
70  }
```

Accumarray3:

```
1   void accumarray3_estado(__global char *subs, int desplazamiento_subs, __constant char *val, int
    nF, int *mResult)
2   {
3       int c0,c1,c2;
4      llenarVector(4*4*4, mResult, 0);
5      int cont = 0;
6       for(int k=0; k<nF*3; k=k+3)
7       {
8           c0 = subs[desplazamiento_subs+k];
9           c1 = subs[desplazamiento_subs+(k+1)];
10          c2 = subs[desplazamiento_subs+(k+2)];
11          //aqui se suma uno en la direccion correspondiente a estas 4 coordenadas en sample
12          //mResult[c0][c1][c2][c3]++;
13          mResult[c0*16+c1*4+c2]+= val[cont];
14          cont++;
15      }
16  }
```

Accumarray4:

```
1   void accumarray4(__global char *subs, int desplazamiento_subs, int nF, int *mResult)
2   {
3       int c0,c1,c2,c3;
4       int cont = 0;
5       llenarVector(256, mResult, 0);
6
7       for(int k=0; k<nF*4; k=k+4)
8       {
9           c0 = subs[desplazamiento_subs+k];
10          c1 = subs[desplazamiento_subs+(k+1)];
11          c2 = subs[desplazamiento_subs+(k+2)];
12          c3 = subs[desplazamiento_subs+(k+3)];
13          //aqui se suma uno en la direccion correspondiente a estas 4 coordenadas en sample
14          //mResult[c0][c1][c2][c3]++;
15          mResult[c0*64+c1*16+c2*4+c3]+= 1;
16          cont++;
17      }
18  }
```

JointEntropy_Accumarray:

```
1  float JointEntropy_accumarray(__global char *matrizA, int desplazamiento_matrizA, int numF, int
   numC)
2  {
3     float hxy2 = 0;
4     float temp_freq;
5      int sample[4*4*4*4];
6
7      accumarray4(matrizA, desplazamiento_matrizA, numF, sample);
8      for(int i=0; i<2; i++)
9      {
10        for(int j=0; j<3; j++)
11        {
12           for(int k=0; k<3; k++)
13           {
14              for(int m=0; m<3; m++)
15              {
16                 if(sample[i*64+j*16+k*4+m] !=0)
17                 {
18                    temp_freq = (float)sample[i*64+j*16+k*4+m]/numF;
19                    hxy2 = hxy2 - temp_freq*log2(temp_freq);
20                 }
21              }
22           }
23        }
24     }
25     return hxy2;
26 }
```

JointEntropy_3loci_accumarray:

```
1  float JointEntropy_3loci_accumarray(__global char *matrizA, int desplazamiento_matrizA, int
   numF,int numC, int *sample3)
2  {
3
4     float hxy2 = 0;
5     float temp_freq;
6      int sample[4*4*4];
7
8      accumarray3(matrizA, desplazamiento_matrizA, numF, sample);//Igual que acumarray4 pero le
   entra todo_ val en unos en vez de el estado
9      //cout<< "terminado" << endl;
10     copiarMatriz3D(sample, sample3, 4);
11
12     for(int i=0; i<3; i++)
13     {
14        for(int j=0; j<3; j++)
15        {
16           for(int k=0; k<3; k++)
17           {
18             if(sample[i*16+j*4+k] !=0)
19             {
20                temp_freq = (float)sample[i*16+j*4+k]/numF;
21                hxy2 = hxy2 - temp_freq*log2(temp_freq);
22             }
23           }
24        }
25     }
26     return hxy2;    //devuelve como parametro de entrada/salida hxy2 y en el return sample3
27 }
```

MutualInfo_Improved:

```
1  float mutualInfo_improved(__global char *matriz, int desplazamiento_matriz, int numF, int numC, int *sample3, __global char *matrizConcSinEstado, int desplazamiento_matrizConcSinEstado)
2  {
3      float FixedEntropyofLabel = 1.0;
4
5      float hxy_father =JointEntropy_accumarray(matriz, desplazamiento_matriz, numF, numC);
6
7      //subset hay que coger la matriz y quitarle el estado
8      rangoDeColumnas(matriz, desplazamiento_matriz, numF, numC, 1, numC-1, matrizConcSinEstado, desplazamiento_matrizConcSinEstado);
9      float hxy_sub2 = JointEntropy_3loci_accumarray(matrizConcSinEstado, desplazamiento_matrizConcSinEstado, numF, numC - 1 ,sample3);//tratar la matriz saltando la primera columna
10     float ixy2=FixedEntropyofLabel+hxy_sub2-hxy_father;
11
12     return ixy2;//ix2y
13 }
```

Kernel GPU:

```
1  __kernel void exhausive_search_GPU(int indice_combinacion, int n_in, int tam_snp_pos_in, int nFil_in, int nCol_in, __global char *matriz, __constant char *estado, __constant float *My_Factorial, __global float *all_score, __global int *all_pos, __global char *snp_com, __global char *mR_concatenarMatrices, __global char *matrizConcatenadaSinEstado, int tamVentana, __global int *vectorCombinaciones, int contadorCombinaciones){
2
3  ////Comienzo Kernel
4      float temp_score1;
5      float temp_score2;
6      float score1;
7      float score2;
8
9      int snp_com_dir_comienzo;
10     int mR_concatenarMatrices_dir_comienzo;
11     int matrizConcatenadaSinEstado_dir_comienzo;
12
13     int j;
14     int k;
15     int WI_id;
16     int numWI_Total;
17     int numComb_x_WI;
18
19     int snp_pos[3];
20
21     snp_com_dir_comienzo = ((int)get_global_id(0))*(tam_snp_pos_in*nFil_in);
22     mR_concatenarMatrices_dir_comienzo = ((int)get_global_id(0))*((1+tam_snp_pos_in) * nFil_in);
23     matrizConcatenadaSinEstado_dir_comienzo = ((int)get_global_id(0))*(tam_snp_pos_in * nFil_in);
```

```
24
25      WI_id = get_global_id(0);
26      numWI_Total = get_global_size(0);
27
28      score1 = 100000;
29      score2 = 100000;
30
31      if(contadorCombinaciones % numWI_Total == 0)
32          numComb_x_WI = contadorCombinaciones / numWI_Total;
33      else
34          numComb_x_WI = (contadorCombinaciones / numWI_Total) + 1;
35
36
37
38      for(int combinacion_actual = WI_id; combinacion_actual<contadorCombinaciones; combinacion_actual = combinacion_actual+numWI_Total)
39      {
40              if(combinacion_actual < contadorCombinaciones)
41              {
42                  snp_pos[0] = vectorCombinaciones[combinacion_actual];
43                  snp_pos[1] = vectorCombinaciones[combinacion_actual+tamVentana];
44                  snp_pos[2] = vectorCombinaciones[combinacion_actual+(2*tamVentana)];
45
46                  separaColumnas(snp_pos, tam_snp_pos_in, matriz, nFil_in, nCol_in, snp_com, snp_com_dir_comienzo);
47
48                  multiscore_obj2(tam_snp_pos_in, snp_com, snp_com_dir_comienzo, nFil_in, tam_snp_pos_in, estado, My_Factorial, &temp_score1, &temp_score2,
49                      mR_concatenarMatrices, mR_concatenarMatrices_dir_comienzo, matrizConcatenadaSinEstado, matrizConcatenadaSinEstado_dir_comienzo,
50                          snp_pos[0],snp_pos[1],snp_pos[2]);
51
52                  all_score[(combinacion_actual)*2] = temp_score1;
53                  all_score[(combinacion_actual)*2+1] = temp_score2;
54                  all_pos[(combinacion_actual)*3] = snp_pos[0];
55                  all_pos[(combinacion_actual)*3+1] = snp_pos[1];
56                  all_pos[(combinacion_actual)*3+2] = snp_pos[2];
57              }
58      }
59      ////Fin Kernel
60  }
```

Kernel CPU:

```
1   __kernel void exhausive_search_CPU(int indice_combinacion, int n_in, int tam_snp_pos_in, int nFil_in, int nCol_in, __global char *matriz, __constant char *estado, __constant float *My_Factorial, __global float *all_score, __global int *all_pos, __global char *snp_com, __global char *mR_concatenarMatrices, __global char *matrizConcatenadaSinEstado, int tamVentana, __global int *vectorCombinaciones, int contadorCombinaciones){
2
3   ////Comienzo Kernel
4       float temp_score1;
5       float temp_score2;
6       float score1;
7       float score2;
8
9       //int snp_pos_dir_comienzo;
10      int snp_com_dir_comienzo;
11      int mR_concatenarMatrices_dir_comienzo;
```

```
12      int matrizConcatenadaSinEstado_dir_comienzo;
13
14      int j;
15      int k;
16      int WI_id;
17      int numWI_Total;
18      int numComb_x_WI;
19
20      int snp_pos[3];
21
22      //snp_pos_dir_comienzo = ((int)get_global_id(0))*tam_snp_pos_in;
23      snp_com_dir_comienzo = ((int)get_global_id(0))*(tam_snp_pos_in*nFil_in);
24      mR_concatenarMatrices_dir_comienzo = ((int)get_global_id(0))*((1+tam_snp_pos_in) * nFil_in);
25      matrizConcatenadaSinEstado_dir_comienzo = ((int)get_global_id(0))*(tam_snp_pos_in * nFil_in);
26
27      WI_id = get_global_id(0);
28      numWI_Total = get_global_size(0);
29
30      score1 = 100000;
31      score2 = 100000;
32
33      if(contadorCombinaciones % numWI_Total == 0)
34          numComb_x_WI = contadorCombinaciones / numWI_Total;
35      else
36          numComb_x_WI = (contadorCombinaciones / numWI_Total) + 1;
37
38      for(int combinacion_actual = WI_id*numComb_x_WI; combinacion_actual<(WI_id*numComb_x_WI)+numComb_x_WI; combinacion_actual++)
39      {
40          if(combinacion_actual < contadorCombinaciones)
41          {
42              snp_pos[0] = vectorCombinaciones[((combinacion_actual)*3)];
43              snp_pos[1] = vectorCombinaciones[((combinacion_actual)*3)+1];
44              snp_pos[2] = vectorCombinaciones[((combinacion_actual)*3)+2];
45
46              separaColumnas(snp_pos, tam_snp_pos_in, matriz, nFil_in, nCol_in, snp_com, snp_com_dir_comienzo);
47
48              multiscore_obj2(tam_snp_pos_in, snp_com, snp_com_dir_comienzo, nFil_in, tam_snp_pos_in, estado, My_Factorial, &temp_score1, &temp_score2,
49                      mR_concatenarMatrices, mR_concatenarMatrices_dir_comienzo, matrizConcatenadaSinEstado, matrizConcatenadaSinEstado_dir_comienzo,
50                          snp_pos[0],snp_pos[1],snp_pos[2]);
51
52              all_score[(combinacion_actual)*2] = temp_score1;
53              all_score[(combinacion_actual)*2+1] = temp_score2;
54              all_pos[(combinacion_actual)*3] = snp_pos[0];
55              all_pos[(combinacion_actual)*3+1] = snp_pos[1];
56              all_pos[(combinacion_actual)*3+2] = snp_pos[2];
57          }
58      }
59      ////Fin Kernel
60  }
```