ISSN 1870-4069

DABATestors un algoritmo basado en CUDA para reducción del tiempo de obtención de testores

Daniel Alejandro Barajas Aranda, Aurora Torres Soto,
María Dolores Torres Soto

Benemérita Universidad Autónoma de Aguascalientes,
México

alengot@hotmail.com

Resumen. El cálculo de testores es un problema de tipo exponencial, por lo que procesar grandes conjuntos de datos para obtener los testores, es una tarea costosa computacionalmente, puede consumir mucho tiempo, llegando a tardar meses en espacios de soluciones de 2^{40}-1. Este trabajo presenta al algoritmo DABATestor, el cual está basado en el paradigma CUDA como una propuesta para la reducción del tiempo de procesamiento. Dicho algoritmo ha sido comparado con un algoritmo secuencial usando matrices de 2x4, 15x7, 57x14, 8421x354, en donde se han obtenido reducciones del orden de una tercera parte del tiempo de procesamiento requerido para la identificación del 100% de los testores de cada matriz. Este algoritmo se ha programado en Python y se pone a disposición de la comunidad científica.

Palabras clave: Testores, CUDA, paralelización, lógica combinatoria, reconocimiento de patrones.

DABATestors an Algorithm Based on CUDA to Reduce Time to Obtain Testors

Abstract. The calculation of testors is an exponential problem, so processing large data sets to obtain them is a computationally expensive, since it is of exponential order and time-consuming task, taking months in 2^{40}-1 solutions' spaces. In this work, the DABATestor algorithm is presented, it is based on the CUDA paradigm, as a proposal to reduce processing time. This algorithm has been compared with a sequential algorithm using 2x4, 15x7, 57x14, 8421x354 matrices, where results of at least one-third of the processing time have been obtained. This algorithm has been programmed in Python and it is put available to the scientific community.

Keywords: Testors, CUDA, parallelization, combinatorial logic, pattern recognition.

1. Introducción

La obtención exhaustiva de los testores de una matriz de datos es una tarea ardua de gran complejidad computacional, que ha sido abordada desde diferentes perspectivas. En este presente artículo, se presenta el algoritmo DABATestors, que está basado en CUDA (Compute Unified Device Architecture - Arquitectura Unificada de Dispositivos de Cómputo). DABATestors, es un algoritmo exhaustivo que funciona con los procesadores nVidia para la obtención en paralelo del 100% de los testores de un grupo de datos.

La solución exhaustiva de gran parte de los problemas del mundo real, es extraordinariamente costosa computacionalmente, como lo es el cálculo de todos los testores asociados a una matriz de aprendizaje. En este documento, se presenta DABATestors, un algoritmo de paralelización determinista de escala exterior que usa el paradigma de programación CUDA para la obtención del conjunto de todos los testores asociados a una matriz de aprendizaje, este algoritmo fue sometido a prueba con matrices de distintos tamaños.

Para evaluar el desempeño del algoritmo DABATestors, se comparó contra un algoritmo secuencial utilizando un GPU Tesla P100-PCIE-16GB con 4 diferentes matrices ejecutando 30 réplicas con cada una de ellas.

1.1. Reconocimiento de patrones

Es una disciplina científica que busca la clasificación de objetos en un número de categorías o clases [1]. Los objetos pueden ser: imágenes, señales, sonidos o cualquier tipo de dato. Desde los años 60's esta disciplina se ha desarrollado gracias al avance de la tecnología, dejando de ser solo una parte teórica de la estadística [1]. Las principales aplicaciones del reconocimiento de patrones son: visión por computadora, reconocimiento de caracteres, diagnóstico asistido por computadora y reconocimiento de voz, entre otras.

La teoría de testores, la cual es una subárea del reconocimiento de patrones, se formuló como una de las direcciones científicas independientes de la cibernética matemática a mediados de los años 50's en la ex-unión de las repúblicas socialistas soviéticas (URSS) [2]. Un testor es un conjunto de características capaz de distinguir objetos provenientes de diferentes clases disjuntas, de manera, que ningún objeto de una clase puede ser confundido con alguno de otra [3].

Los testores han sido aplicados en varios ámbitos, por ejemplo: en medicina se empleó en la determinación de factores asociados con lesiones pulmonares agudas relacionadas con transfusiones sanguíneas (TRALI); se aplicaron mediante un algoritmo evolutivo híbrido [4]. También se han aplicado en astronomía para la estimación de parámetros estelares [5], así como en ciencias computacionales, donde han sido utilizados para reducir las dimensiones de modelos de redes neuronales [6]. Los testores son una herramienta muy útil, sin embargo, su cálculo es un proceso tardado que crece exponencialmente con cada una de las variables que tenga la matriz a procesar [7].

Fig. 1. CPU vs GPU [10].

Para el lector interesado en profundizar sobre testores y los conceptos relacionados con esto, se recomienda consultar a Schucloper (Introducción a la Teoría de Testores J Ruíz-Shulcloper, E Alba, M Lazo - Serie Verde Cinvestav-IPN, 1994).

1.2. Cómputo paralelo

La evolución que han tenido las tarjetas gráficas ha hecho factible el empleo de dos o más procesadores para la solución computacional de un problema, mediante su división en componentes independientes que pueden resolverse de manera paralela.

Lo anterior, bajo el principio de que los grandes problemas se pueden dividir en otros más pequeños [8].

Entre las diferentes razones que existen para el uso del cómputo en paralelo, se pueden destacar tres [9]:

- La transformación y creación de algoritmos de modo paralelo para resolución de problemas.
- La capacidad del cómputo paralelo de disminuir el tiempo de cómputo necesario en problemas de grandes volúmenes de datos o cálculos intensivos.
- El límite físico alcanzado por las computadoras secuenciales.

1.3. CUDA

CUDA son las siglas de Arquitectura Unificada de Dispositivos de Cómputo (Compute Unified Device Architecture), que es una plataforma de computación en paralelo y un conjunto de herramientas de desarrollo, creadas por nVidia que permiten la codificación en GPU (graphics processing unit). En la Figura 1 se muestra que un GPU contiene más núcleos ALU (arithmetic logic unit- unidad aritmética lógica) que un CPU y que éstos comparten la memoria, lo que funciona conceptualmente para cálculos altamente paralelos porque la GPU puede ocultar las latencias de acceso a la memoria en lugar de acceder a ella a través de grandes cantidades de memoria caché.

Dentro de los GPU nVidia, existe una jerarquía de subprocesos, cada bloque de subprocesos puede programarse en cualquiera de los multiprocesadores disponibles dentro de una GPU, en cualquier orden, de forma simultánea o secuencial, de modo que un programa CUDA compilado pueda ejecutarse en cualquier número de multiprocesadores como se ilustra en la Figura 2 [10]. En esta arquitectura el programador solo tiene que definir los bloques de procesos, y automáticamente el GPU se encarga de asignarlos a los núcleos que tenga disponibles.

Tabla 1. Matrices de Prueba.

Matriz	Registros	Variables	Casos estudiados del Conjunto Potencia
M1	57	14	16384
M2	8421	354	100000, 10000000
M3	2	4	16
M4	15	7	128

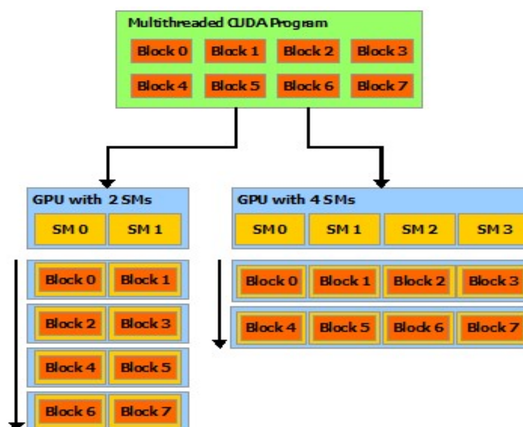

Fig. 2. Escalabilidad Automática [10].

Un programa multiproceso, se divide en bloques de subprocesos que se ejecutan independientemente uno del otro, de modo que una GPU con más procesadores ejecutará automáticamente el programa en menos tiempo que una GPU con menos procesadores [10].

2. Materiales y métodos

El algoritmo DABATestors fue implementado bajo el paradigma CUDA en el lenguaje Python. Este requiere de la librería numba y el uso de CUDA Toolkit versión 8 o superior para su correcto funcionamiento. Adicionalmente, se utilizaron las librerías numpy y pandas para el uso de matrices.

El algoritmo recibe como insumos una matriz básica que contiene la información sintetizada sin los superconjuntos de la matriz de diferencias, y un conjunto potencia el cual está formado por todas las combinaciones binarias del número de variables que componen el problema: 2^n-1 posibilidades, donde **n** es el número de variables que contiene el problema.

Cabe resaltar que la matriz básica se debe tener previamente en memoria, mientras que la matriz del conjunto potencia se define en tiempo de ejecución. La implementación del algoritmo consiste en establecer el proceso general para la obtención de un solo testor, y este proceso se paraleliza en varios hilos pertenecientes a varios bloques en la

```
Definir función (IsTestor)
        Identificar hilo de GPU
        Obtener número de testor guardar en decimal
        Mientras decimal // !=0
                Arreglo1 en [-j]= decimal % 2
                Incrementar j
        Arreglo1 [-j] = decimal
        Band=1
        Bucle hasta total de filas de matriz básica (j)
                Band2=0
                Bucle hasta total de columnas de matriz básica (k)
                Si arreglo1 [k] ==1
                        Si matriz básica [j,k]==1
                                Band2++
                Si band2==0
                        Band1=0
        Si band1 ==1
                Retornar true
        Si no
                Retornar false
Definir dimensión de malla y bloques
Definir tamaño de paso
Llamar función con arreglo del tamaño de paso empezando desde 0
Si último número llamado < 2^(número de columnas matriz básica)
        Tamaño de paso *2
        Llamar función con arreglo del tamaño de paso empezando en último número llamado
Repetir desde el si
```

Fig. 3. Pseudocódigo del Algoritmo DABATestors.

cuadrícula del GPU. El algoritmo se puso a prueba con diversas matrices (Tabla 1) obteniendo los testores para problemas conocidos con anterioridad (M1, M2), donde se trabaja datos de personas con tendencia suicida [11]; explorando el espacio total de soluciones en la matriz M1; y en la matriz M2 se verificaron 100,000 y 10,000,000 de casos del conjunto potencia. También se usaron matrices M3 y M4 en donde se exploró el espacio completo de soluciones.

La evaluación del algoritmo se llevó a cabo en un GPU Tesla P100-PCIE-16GB, en donde se ejecutó 30 veces para cada matriz. El algoritmo secuencial también fue replicado 30 veces. El pseudocódigo de la herramienta desarrollada (bajo el paradigma CUDA), se muestra a continuación (Fig. 3).

Se puede apreciar que el pseudocódigo define el tamaño de bloques y mallas a trabajar por el GPU, y define el tamaño de paso (tamaño del subconjunto del conjunto potencia) a utilizar en la iteración, enseguida llama a la función IsTestor enviando subconjuntos del conjunto potencia. En la función identifica el hilo utilizado por el GPU para tomar el registro del conjunto potencia y verificar si es o no testor, transformando el número a binario y posteriormente comparándolo con la matriz básica.

3. Resultados

Se obtuvo el tiempo de procesamiento de las diversas matrices mediante el algoritmo DABATestors y un algoritmo secuencial que utiliza la misma función descrita en el pseudocódigo, pero de forma secuencial. Mediante la comparación de los tiempos de

```
@cuda.jit
def testor(mb,mto,bina,col,fil,lim): #matriz basica, matriz de testores
                  #y salida, vector de binario, columnas, filas, limite
  i = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

 #obtención del número a verificar testor
  decimal=mto[i,0]

#crear binario
  j=1
  while decimal // 2 != 0:
    bina[i,-j]  = (decimal % 2)
    decimal = decimal // 2
    j+=1
  bina[i,-j]=decimal

#verificar is testor con la matriz básica
  istestor=1
  for j in range(fil):
    cad0=0
    for k in range (col):
      if bina[i,k]==1:
        if mb[j,k]==1:
          cad0=cad0+1
    if cad0==0:
      istestor=0

    if istestor!=0:
      mto[i,1]=1
    else:
      mto[i,1]=0
#fin de la función paralela

#defición del bloque de hilos
griddim = 1000, 1
blockdim = 1000, 1
i=0
paso=1000000

#creación de los arreglos
mto=numpy.arange((paso*i), (paso*(i+1)), 1, dtype=numpy.int32)
mto=mto.reshape((paso, 1))
mto=numpy.insert(mto, 1, 0, axis=1)
bina = numpy.zeros((paso, col), dtype=numpy.int32)
mto_out = cuda.device_array_like(mto)

mto_out[:]=mto[:]
lim=2**col
lim=0
mto_out.copy_to_host(mto)

#ciclo de ejecución para obtención de testores
for j in range(200):
  mto=numpy.arange((paso*i), (paso*(i+1)), 1, dtype=numpy.int32)
  mto=mto.reshape((paso, 1))
  mto=numpy.insert(mto, 1, 0, axis=1)
#llamada a la función paralela
  testor[griddim, blockdim](mb,mto_out,bina,col,fil,lim)

  mte=pd.DataFrame(mto)
  mte=mte[mte[1] == 1]
```

Fig. 4. Algoritmo DABATestors.

procesamiento, entre el algoritmo DABATestors y el secuencial, se estableció el punto donde el uso del algoritmo DABATestors es similar que uno secuencial (conjunto potencia de (2^7)-1). Uno de los resultados más importantes de este trabajo es el código de la herramienta DABATestors (Fig. 4). Así como la comprobación de su efectividad.

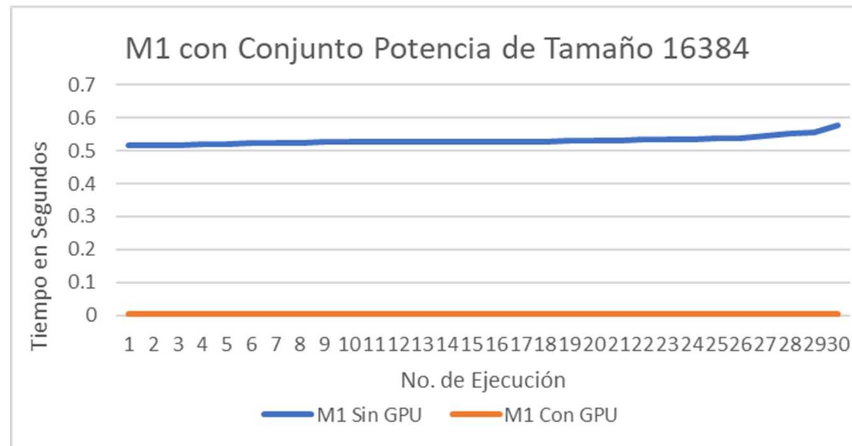

Fig. 7. Tiempo de procesamiento de la matriz M1 con 16384 casos del conjunto potencia.

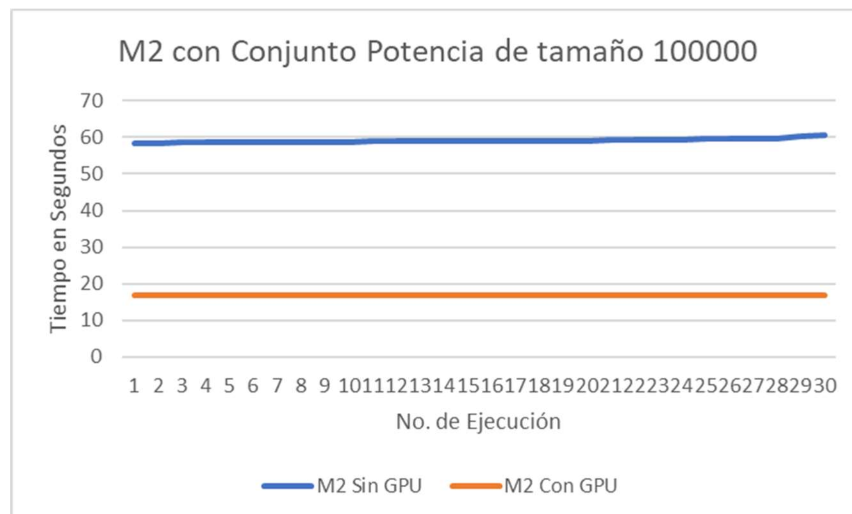

Fig. 8. Tiempo de procesamiento de la matriz M2 con 100,000 casos del conjunto potencia.

En la figura 4 se puede observar que el tamaño de paso es de 1,000,000, esto se define de esta manera ya que el procesador utilizado no soporta un número mayor de datos en paralelo, pudiendo aumentar o disminuir este número según el procesador disponible. Se realizaron pruebas de desempeño en las que se midió el tiempo de ejecución durante 30 réplicas.

La función testor descrita en el algoritmo se ejecutó de manera secuencial y paralela mediante CUDA en un GPU Tesla P100-PCIE-16GB, obteniendo mediciones para las matrices M1, M2, M3 y M4 mencionadas previamente.

En la figura 5 se observa el tiempo promedio de las pruebas efectuadas en los diferentes casos. En el procesamiento de casos del conjunto potencia (CP) de tamaño 16, 6384 y 100,000 filas respectivamente el tiempo de procesamiento con CUDA (línea naranja)

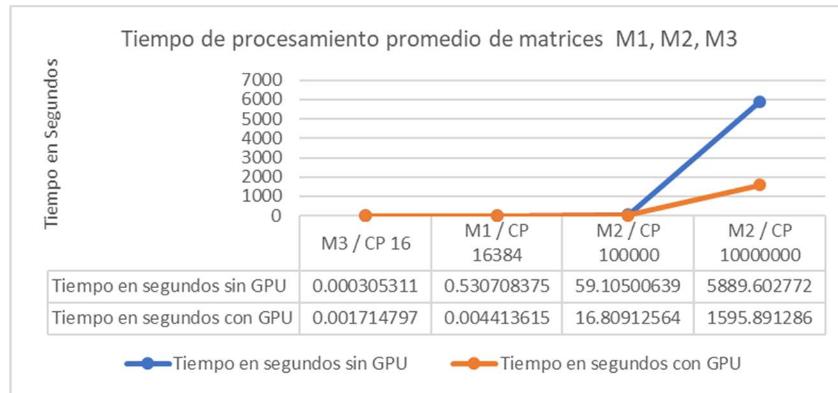

Fig. 5. Tiempo de procesamiento promedio de matrices M3, M1, M2.

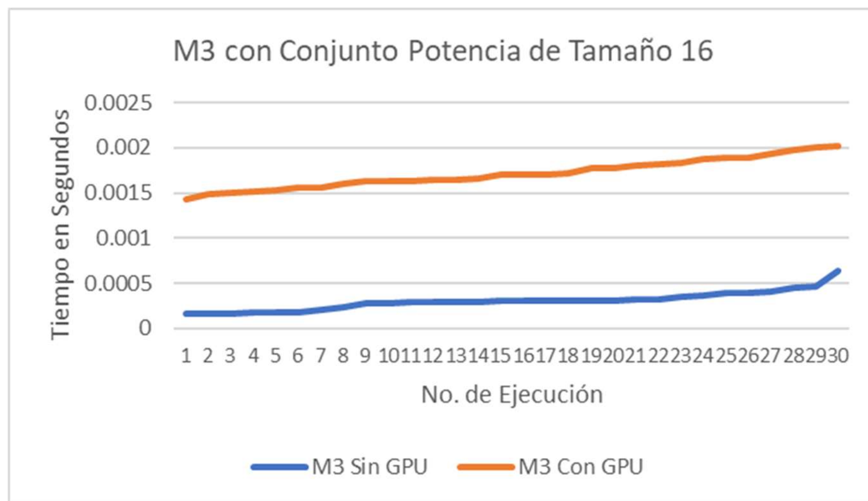

Fig. 6. Tiempo de procesamiento de la matriz M3 con 16 casos del conjunto potencia.

aparenta ser similar a el procesamiento secuencial, sin embargo, cuando el espacio de soluciones es grande (10,000,000), se puede apreciar un desempeño muy bueno en el procesamiento con CUDA.

En la figura 6 se observa que el uso del algoritmo DABATestors no es una buena opción cuando se tiene un grupo de datos pequeño, ya que el tiempo de procesamiento fue mayor al requerido en el procesamiento secuencial (línea azul).

Esto ocurrió debido a que los tiempos requeridos para cargar los datos en memoria son mayores que el tiempo que tarda en procesarlos.

En las figuras 7 y 8, se observa la diferencia entre el tiempo de procesamiento, entre el algoritmo DABATestors y el secuencial. Se puede apreciar que los tiempos requeridos con el uso de GPU (línea naranja) son significativamente inferiores a los requeridos sin el uso de este (línea azul), mostrando resultados de al menos un tercio del tiempo requerido por un algoritmo secuencial. En el caso de la figura 9 se realizó el

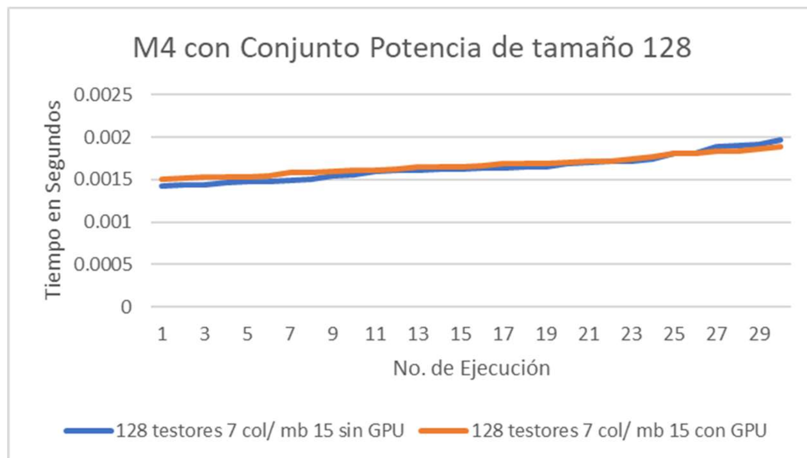

Fig. 9. Tiempo de procesamiento de la Matriz M4 con 128 casos del conjunto.

análisis de la matriz M4 de sus 128 testores posibles. Se observa que el tiempo de procesamiento con el uso del GPU (línea naranja) y sin él (línea azul) son muy semejantes. La grafica muestra los resultados de 30 ejecuciones diferentes, donde se obtuvieron tiempos muy similares, mostrando que es conveniente dejar de usar el procesamiento secuencial y empezar a usar el procesamiento en paralelo en matrices de características similares a la matriz M4 cuyo conjunto potencia es de (2^7)-1 filas, sin importar en gran medida el número de registros de la matriz básica.

Del grupo de experimentos expuestos se observa que cuando el conjunto potencia crece (a partir de (2^7)-1) filas, el uso del algoritmo DABATestors es una opción muy buena, ya que se reduce el tiempo de procesamiento. Con el algoritmo DABATestors se mejoró significativamente el tiempo de procesamiento logrando reducirlo a un tercio del requerido por su contraparte secuencial en problemas con conjuntos potencia de al menos (2^{14})-1.

4. Conclusiones y trabajo a futuro

Los problemas del mundo real raramente son abordados con técnicas exhaustivas debido a su alto costo computacional; un buen ejemplo es la identificación del 100% de los testores asociados a una matriz. En este trabajo, se presenta una alternativa que logra reducir el tiempo de procesamiento requerido por un algoritmo exhaustivo y secuencial en al menos una tercera parte para problemas con conjuntos potencia de (2^{14})-1 filas por lo bajo.

Se ha demostrado que el uso del algoritmo DABATestors provee resultados muy convenientes en cuanto al tiempo de ejecución, no obstante, con problemas con pocas variables, es mejor usar una técnica secuencial; debido al tiempo requerido en la copia de datos a la memoria del GPU, sin embargo, cuando el tiempo de procesamiento es superior al tiempo de transferencia de datos a la memoria, el uso del algoritmo DABATestors representa una mejor alternativa que un algoritmo secuencial.

Cuando los problemas tienen una complejidad algorítmica muy elevada y se requiere que la solución sea determinista, el uso del GPU es una buena alternativa, ya que permite reducir el tiempo de procesamiento sustancialmente. En el caso de la obtención de los testores, el algoritmo propuesto brinda una reducción del tiempo de procesamiento, requiriendo un tercio del tiempo del algoritmo secuencial.

En este trabajo se presenta un algoritmo que está a disposición de la comunidad científica. Los resultados presentados se obtuvieron con un GPU Tesla P100-PCIE-16GB con diversas matrices obteniendo mejoras de tiempo significativas en matrices básicas de al menos 14 columnas.

Actualmente se está trabajando en la obtención del total de testores típicos como parte de una nueva propuesta de este algoritmo.

Referencias

1. Prandi, D., Gauthier, J. P.: A semidiscrete version of tehe Citti-Petitot-Sarti model as a plausible model for anthropomorphic image reconstruction and pattern recognition, Springer Briefs in Mathematics (2018)
2. Dmitriev, A. N., Zhuravlev, Y. I., Krendeleiev, F. P.: On mathematical principles of object and phenomena classification, Discrete Analysis, pp.3–15 (1966)
3. Shulcloper, J. R., Guzmán, A., Martínez, J. F.: Enfoque lógico combinatorio al reconocimiento de patrones. Instituto Politécnico Nacional, Mexico (1999)
4. Torres, M. D., Torres, A., Cuellar, F., Torres, M. L., Ponce de León, E., Pinales, F.: Evolutionary computation in the identification of risk factors. Case of TRALI (2014)
5. Santos, J. Á., Carrasco, A., Martínez, J. F.: Selección de características usando testores típicos aplicada a la estimación de parámetros estelares. Computación y Sistemas vol. 8, pp. 15–23 (2004)
6. Vázquez, R A., Godoy-Calderón, S.: Using testor theory to reduce the dimension of neural network models. Spec Issue Neural Networks Assoc Memories vol. 28, pp. 93–103 (2007)
7. Alganza, Y. S., Porrata, A. P.: Lex: Un nuevo algoritmo para el calculo de los testores tipicos. Ciencias Matemáticas, vol. 21, no. 1 (2003)
8. Almasi, G. S.: Highly parallel computing. Benjamin/Cummings series in computer science and engineering (1989)
9. De Giusti, A., Naiouf, M., Sanz, C., Tinetti, F., De Giusti, L, Bertone, R.: Procesamiento paralelo y distribuido en tratamiento masivo de datos. Un workshop de investigadores en ciencias de la computación, pp. 364-367 (2002)
10. Programming Guide : CUDA toolkit documentation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction. Accessed 26 Mar 2020
11. Barajas, D.: Identificación de factores de riesgo determinantes en el suicidio en Aguascalientes mediante la técnica de testores típicos. Universidad Autónoma de Aguascalites (2017)