Instituto Politécnico Nacional

Centro de Investigación en Computación

Implementación de algoritmos de agrupamiento en CUDA

T E S I S

Que para obtener el grado de:

Maestría en Ciencias de la Computación

P R E S E N T A :

Ing. Verónica Patricia Ávila Rubio

Director(es) de Tesis:

Dr. Ricardo Menchaca Méndez

Dr. Rolando Menchaca Méndez

DICIEMBRE 2016

SIP-14 bis

INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

ACTA DE REVISIÓN DE TESIS

En la Ciudad de México siendo las 16:00 horas del día 12 del mes de diciembre de 2016 se reunieron los miembros de la Comisión Revisora de la Tesis, designada por el Colegio de Profesores de Estudios de Posgrado e Investigación del:

Centro de Investigación en Computación

para examinar la tesis titulada:

"Implementación de algoritmos de agrupamiento en CUDA"

Presentada por el alumno:

ÁVILA	**RUBIO**	**VERÓNICA PATRICIA**
Apellido paterno	Apellido materno	Nombre(s)

Con registro:

B	1	4	0	4	5	2

aspirante de: **MAESTRÍA EN CIENCIAS DE LA COMPUTACIÓN**

Después de intercambiar opiniones los miembros de la Comisión manifestaron ***APROBAR LA TESIS***, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

LA COMISIÓN REVISORA

Directores de Tesis

Dr. Ricardo Menchaca Méndez

Dr. Rolando Menchaca Méndez

Dr. Ricardo Barrón Fernández

Dr. Francisco Hiram Calvo Castro

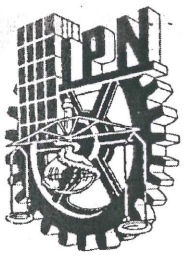
Dr. Miguel Jesús Torres Ruiz

Dr. José Giovanni Guzmán Lugo

PRESIDENTE DEL COLEGIO DE PROFESORES

Dr. Marco Antonio Ramírez Salinas

INSTITUTO POLITECNICO NACIONAL
CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN
DIRECCIÓN



INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

CARTA CESIÓN DE DERECHOS

En la Ciudad de México el día 12 del mes de diciembre del año 2016, la que suscribe Ing. Verónica Patricia Ávila Rubio alumna del Programa de Maestría en Ciencias de la Computación con número de registro B140452, adscrito al Centro de Investigación en Computación, manifiesta que es autora intelectual del presente trabajo de Tesis bajo la dirección de Dr. Ricardo Menchaca Méndez y Dr. Rolando Menchaca Méndez y cede los derechos del trabajo intitulado "Implementación de algoritmos de agrupamiento en CUDA", al Instituto Politécnico Nacional para su difusión, con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y/o director del trabajo. Este puede ser obtenido escribiendo a la siguiente dirección: veronica.vpar@gmail.com. Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

Verónica Patricia Ávila Rubio

Nombre y firma

Resumen

Con el avance cada vez más rápido de la tecnología, una gran parte de las aplicaciones generan una cantidad cada vez más grande de datos a gran velocidad; datos que sin una interpretación carecen de una utilidad real; por ello es que el análisis de datos se ha convertido en un método común tanto en la industria como en la investigación científica moderna; lo que nos pone frente al reto de mejorar los algoritmos o procedimientos de análisis tradicionales. Es aquí donde campos de investigación como aprendizaje de máquina (Machine Learning) aportan soluciones tales como los algoritmos de clasificación utilizados principalmente como métodos de aprendizaje supervisado y algoritmos de clustering (agrupamiento de datos) utilizados en su mayoría como métodos de aprendizaje no supervisado. Dichos algoritmos al enfrentarse a las nuevas características de los datos incrementan su complejidad computacional.

Una de las alternativas de solución es implementar dichos algoritmos en esquemas de computación paralela. Dentro del siguiente trabajo se presenta un análisis realizado sobre las características con las que un algoritmo de aprendizaje máquina debe contar para ser implementado en CUDA con mayor éxito, además de un análisis sobre las ventajas y desventajas que ofrece CUDA frente a otros modelos de programación paralela. A través de puntualizar las características de los cuatro tipos de computación paralela (paralelismos a nivel bit, a nivel de instrucción, a nivel de datos, a nivel de tareas) se intenta obtener una generalización que nos ayude a definir cuándo y con que tipo de programación paralela es recomendable implementar un algoritmo de aprendizaje máquina.

Se realiza también una comparación entre los tiempos de ejecución y el performance que presenta el algoritmo Mezcla de Gaussianas, viéndolo como un caso particular del algoritmo EM (Expectation-Maximization), implementado en serie con Python y en paralelo con CUDA.

Abstract

In the last decades the immense growth of data has been come a driving force to develop alternatives to improve existing methods and to get better performance to solve everyday problems for companies and various research areas such as genetics, or geospatial processing. Machine learning algorithms have been adapted to better cope with the mass of data being processed. In this topic clustering has been a great solution for the science. Clustering is a common unsupervised learning technique used to discover group structure in a set of data. A cluster is a collection of objects which are similar between them and dissimilar to the objects belonging to other clusters. There are many clustering algorithms in the literature; several optimization techniques lead to improvements in performance and scalability among which parallelization is one valuable option.

One of the alternative solutions is to implement such algorithms in parallel computing schemes, in the following work an analysis is presented on the characteristics of machine learning algorithms should have to be implemented in CUDA with greater success. Additionally, an analysis on the advantages and disadvantages of CUDA respect to the cluster programming is presented; through specifying the characteristics of the four types of parallel computing (bit level, instructional level, data level, task level) we try to obtain a generalization that helps us to define when and with what kind of parallel programming it is advisable to implement a machine learning algorithm.

Also a comparision between the execution times and performances of the Mixtures of Gaussians algorithm, as a particular case of the Expetation-Maximization algorithm, is presented. Implemented serial and parallel versions with languages programming, python and CUDA respectively.

Agradecimientos

Gracias a mis padres Francisco Ávila y Patricia Rubio, a mis hermanas Marisela y Luz, a la mejor abuelita que la vida me pudo dar María González y a mi tío Pedro Rubio; por su apoyo, la confianza, por soportar de mí los peores momentos, y por todo su cariño. Todo lo que soy es por y para ustedes, los amo.

Gracias a mis asesores Ricardo Menchaca Méndez y Rolando Menchaca Méndez por el tiempo y el conocimiento transmitido, me llevo mucho más de lo que puedo demostrar de ustedes.

Gracias a mis amigos José Aguilar y Alexander Alvarado por todo lo que engordamos, platicamos y vivimos juntos, siempre vivirán en mi corazón.

Gracias a mis amigos Mario Aburto, Ángel Mandujano, Pabel Carrillo y Jessica De Anda por los *mil y un clubs* y todas las pláticas vespertinas.

Gracias a mis amigas *las innombrables* por su apoyo, su cariño y por siempre poner una sonrisa en mi rostro; Abi, Angie, Au, Dany, Eli, Martha, Moni ¡gracias familia!

Gracias a Sofí Segundo, a Nancy Blanco y a Magdalena Saldaña por sacar lo mejor de mi persona.

Gracias a Celi Mendoza y Jazmín Ávila, por su apoyo y enseñanzas.

Gracias al CIC, al CONACyT y sobre todo gracias a mi alma máter el Instituto Politécnico Nacional.

Dedicatoria

A quién corresponda.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Con el avance cada vez más rápido de la tecnología, una gran parte de las aplicaciones generan una cantidad cada vez más grande de datos a gran velocidad (ejemplo: redes de tráfico, redes de sensores); datos que sin una interpretación carecen de una utilidad real; por ello es que el análisis de datos se ha convertido en un método común tanto en la industria como en la investigación científica moderna; lo que nos pone frente al reto de mejorar los algoritmos o procedimientos de análisis tradicionales. Es aquí donde campos de investigación como aprendizaje de máquina (Machine Learning) aportan soluciones tales como los algoritmos de clasificación utilizados principalmente como métodos de aprendizaje supervisado, mismo que se presenta cuando se conoce a priori el resultado final óptimo del algoritmo, y algoritmos de clustering (agrupamiento de datos) utilizados en su mayoría como métodos de aprendizaje no supervisado, es decir métodos donde se desconoce el resultado final. Dichos algoritmos al enfrentarse a las nuevas características de los datos incrementan su complejidad computacional.

En el campo de la ciencia de la computación los problemas tratados suelen ser clasificados en función de su complejidad computacional, la cual puede ser medida en términos de espacio y tiempo. Es decir, en función de su complejidad espacial (cantidad de espacio o cantidad de memoria requerida) y en función de su complejidad temporal (la cantidad de tiempo que es requerido para su resolución). Las principales clases de complejidad temporal en que se clasifican los problemas computacionales son la polinomial (P), polinomial no determinista (NP), polinomial no determinista completa (NP-Complete) y la polinomial no determinista difícil (NP-Hard). Podemos definir a la clase P como la clase en la que se encuentran todos los problemas que pueden ser resueltos en tiempo polinomial respecto a su entrada, dicha clase es un subconjunto de la clase NP. La clase NP está conformada

por los problemas para los cuales no necesariamente se conoce un algoritmo que pueda resolverlos eficientemente, pero que sí pueden ser verificados en tiempo polinomial. Un problema es verificable en tiempo polinomial cuando, dada una entrada y una supuesta solución es posible determinar en tiempo polinomial si dicha solución es una solución válida al problema. [24]

Agrupamiento es uno de los más famosos problemas de aprendizaje no supervisado, su objetivo principal es de carácter descriptivo, es decir busca la estructura que mejor represente a la información, y consiste en crear grupos a partir de un conjunto de datos que se le presenta, donde los integrantes de cada grupo compartan características y al mismo tiempo sean diferentes a los datos que se encuentran en grupos distintos al suyo. En algunos problemas de agrupamiento el número de clústeres o grupos que son formados, K, es dado como un parámetro; en algunos otros (K) es desconocido. Existe una amplia gama de algoritmos de agrupamiento cada uno con sus propias fortalezas y debilidades mismas que son descritas por la complejidad de la información que procesen.

Agrupamiento se ha empleado en muchas disciplinas tales como la estadística, la biología y la medicina principalmente con tres propósitos: [22]

- Descubrir la estructura de los datos,
- Encontrar la clasificación natural de los objetos de datos mediante la identificación de similitudes entre ellos, y
- Organizar y resumir los datos.

En algunas ocasiones el agrupamiento puede ser utilizado como una herramienta para pre-procesa-miento de los datos para otras tareas, tales como la regresión, análisis de componentes principales, análisis de asociación, incluso podemos citar como ejemplo trabajos en los que los autores utilizan técnicas de agrupamiento para reducir la complejidad en la solución de problemas de planificación automatizados (automated planning problems) [14]. Típicamente los problemas de agrupamiento se consideran como problemas de optimización con objetivos distintos y resueltos por algoritmos exactos, métodos de aproximación o métodos heurísticos.

Dentro de la literatura existen diversas clasificaciones de los algoritmos de agrupamiento; la más común es dividirlos en cuatro tipos: agrupamiento jerárquico, agrupamiento por

particiones, agrupamiento basado en densidad y agrupamiento basado en grid [2]; sin embargo dentro del estado del arte es posible encontrar clasificaciones más extensas para los algoritmos tradicionales y modernos de agrupamiento misma que será revisada durante futuros capítulos del presente trabajo.

A lo largo del tiempo expertos en aprendizaje máquina y probabilidad se han percatado de que el análisis de agrupamiento también puede basarse en modelos de probabilidad y han apostado por trabajar en conjunto para mejorar algoritmos de aprendizaje automático con tendencias probabilistas, en agrupamiento específicamente la probabilidad brinda la oportunidad de crear modelos que puedan describir la estructura de los datos aunque no de forma perfecta sí de una forma efectiva, sin la necesidad de tener un conocimiento previo sobre dicha estructura. Un ejemplo de ello es el modelo de mezclas (mixtures model) y lo podemos definir como un modelo probabilístico para representar la presencia de subpoblaciones dentro de una misma población; es decir una distribución de probabilidad conformada por una superposición lineal de varias distribución de probabilidad que caracteriza a un conjunto de datos. Donde cada una de las distribuciones de probabilidad pueden representar a cada uno de los clústeres que se pueden formar dentro del conjunto de datos que se trabaja. Si decimos que estas distribuciones son gaussianas hablamos de un modelo de mezcla de Gaussianas (mixtures of Gaussians), mismo que se encuentra basado en el modelo de distribución de probabilidad del mismo nombre.

1.1. Planteamiento del problema

La generación de grandes cantidades de datos se considera una constante cuando de nuevo conocimiento, o nuevas formas de tecnología hablamos; como ya se mencionó, dicha cantidad carece de valor sin un correcto procesamiento que nos permita extraer toda la información que pueda ofrecernos. A pesar de que a diario expertos en el ramo desarrollan nuevo hardware que permitan hacer que el procesamiento de datos resulte eficiente, eficaz, y sobre todo lo más rápido que sea posible, la realidad es que el ritmo al que se genera nueva información nos sigue rebasando. Aunque almacenar grandes cantidades de información no es algo que resulte ser nuevo, pues desde hace mucho tiempo las empresas han ido guardando información acerca de sus clientes, proveedores y operaciones creando así grandes bases de datos, de igual forma dentro del campo científico se han logrado importantes avances, descubrimientos vertidos en extensos repositorios de información. Hace no mucho, nos dimos cuenta de que todos esos datos con sus individualidades nos generan conocimiento tan significativo que puede ayudarnos en la toma de decisiones que trasciendan por sus efectos.

Por nombrar un ejemplo mencionaré los repositorios con la información de la estructura del ADN para estudiar posibles mutaciones que pueden ser benéficas para la aparición de cáncer, se han logrado obtener bastante información al respecto pero, ¿cómo utilizarla de forma que resulte en nuestro beneficio? o en el mundo empresarial ¿cómo utilizar la información sobre clientes y/o usuarios para obtener beneficios? se apuesta por predecir el comportamiento de dichos factores, continuando con los ejemplos, a partir de alteraciones que presente una estructura ADN y con base en las observaciones con las que se cuenta se trata de predecir que estructuras son propensas a la aparición de cáncer. Lo que se busca detectar a tiempo esta enfermedad y poder darle una solución que salve la vida de muchas personas. Al hablar de la información de una empresa, lo de hoy es a través de conocer las preferencias y gustos del cliente ofrecerle servicios especializados o identificar en qué tipo de mercado es más factible lanzar un producto o servicio a fin de que el consumo sea un éxito.

El problema computacional gira en torno a la cantidad de información, a los recursos tecnológicos con los que contamos para poder llevar a cabo tareas como las ya mencionadas y a la velocidad en que se requieren respuestas. El tiempo de procesamiento que requiere un algoritmo se encuentra en función del tamaño de la entrada (cantidad de datos) que se tenga y, como se ha mencionado brevemente, uno de los objetivos que se buscan en el campo de la computación es crear algoritmos eficaces que puedan resolver problemas difíciles en un tiempo considerable es decir en tiempo polinomial. Es por ello que se busca explorar una de las opciones más viables que hasta el momento se conocen: el cómputo paralelo, para con esto disminuir el tiempo de procesamiento y hacer eficientes algoritmos que ya son conocidos pero poco utilizados para así utilizar las ventajas que estos ofrecen al realizar tareas de clasificación dentro de campos de la ciencia y la industria o en donde dichas tareas puedan resultar de gran utilidad.

1.2. Objetivos

1.2.1. Objetivo General

Implementar algoritmos de agrupamiento, de uso poco común, en cómputo paralelo, utilizando CUDA.

1.2.2. Objetivos específicos

1. Conocer algoritmos clásicos de agrupamiento en aprendizaje máquina.
2. Implementar algoritmos clásicos de agrupamiento en algún lenguaje de programación.
3. Conocer y trabajar con arquitecturas paralelas.
4. Conocer y trabajar con la arquitectura CUDA.
5. Definir qué tipo de algoritmos se pueden implementar en CUDA.
6. Realizar un análisis sobre el performance de la implementación del algoritmo Mezcla de Gaussianas en CUDA.

1.3. Justificación

Dentro del ámbito del aprendizaje de máquina existe una gran cantidad de algoritmos con un alto rendimiento y precisión en la tarea de clasificar datos y/o predecir información; sin embargo la complejidad de dichos algoritmos hace que el implementarlos de forma serial sea muy poca dentro de las distintas áreas de estudios. Un ejemplo de ello son los algoritmos basados en distribuciones de probabilidad, incluso los algoritmos aleatorios, que se basan en operaciones matriciales, las cuales pueden ser hasta de grado igual al número de rasgos de los datos.

Hoy en día existen distintas alternativas cuando de cómputo paralelo se habla. En cuanto al hardware se han desarrollado distintos modelos que permiten a las personas especialistas en la materia implementar software de gran complejidad y con esto reducir el tiempo de ejecución manteniendo el performance.

Al momento de elegir entre las opciones de arquitecturas paralelas, se encuentra útil realizar un análisis en cuanto a las ventajas y desventajas que nos ofrece una arquitectura como CUDA y alguna otra alternativa de computo paralelo, como lo es mapReduce; a su vez puede resultar de gran utilidad encontrar en que proporción CUDA (en específico) acelera el procesamiento de algoritmos como mezcla de gaussianas.

Capítulo 2

Marco Teórico

Durante el siguiente capítulo se presentan algunos conceptos que sirven como fundamento para la presente tesis.

2.1. Aprendizaje de Máquina

Aprendizaje de máquina (Machine Learning) es un concepto que suele estar fuertemente relacionado con Inteligencia Artificial (IA), incluso es considerada por muchos como una rama de dicha ciencia; sin embargo el aprendizaje de máquina se centra en aplicaciones más prácticas como [25] menciona: la tarea de enseñar a una máquina a aprender está más ligada con un problema en específico, es decir, el aprendizaje de máquina es más parecido al entrenamiento de un empleado que a la tarea de criar a un niño.

La primera definición de aprendizaje de máquina se atribuye a Arthur Samuel, quien en 1959 lo define como: "campo de estudio que proporciona a las computadoras la habilidad de aprender sin ser programada de forma explicita" [23]. Una definición más reciente es la de Tom Mitchell en [32]: "Se dice que un programa informático aprende de la experiencia E con respecto a alguna tarea T y a alguna medida de rendimiento P, si, su rendimiento en T, medido por P, mejora con la experiencia E."

> "A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E." [32]

El objetivo del aprendizaje de máquina es crear sistemas que sean capaces de cambiar su comportamiento de manera autónoma con base en la experiencia que adquieren [25], o

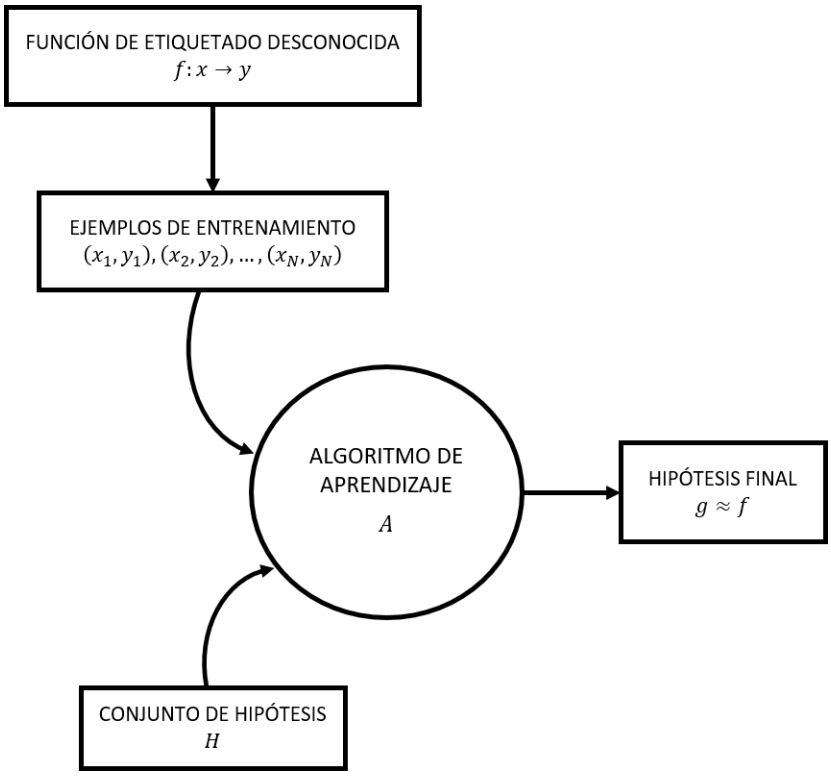


Figura 2.1: Representación gráfica de un problema de aprendizaje.

dicho de otra forma, sistemas con la capacidad de predecir futuros eventos o escenarios desconocidos para la computadora, basándose sólo en los datos que se le presentan. Aprender de los datos comienza a ser útil en situaciones donde no hay una solución analítica a priori, pero se cuenta con datos que pueden ser utilizados para construir una solución empírica.

Es así que el aprendizaje de máquina está enfocado a resolver problemas de conocimiento compuestos por: un conjunto de datos de entrada $\mathbf{X}$, una función desconocida de etiquetado $f : \mathtt{X} \rightarrow \mathtt{Y}$ donde $\mathbf{X}$ es el espacio de entrada (conjunto de las posibles entradas $\mathbf{x}$), $\mathbf{Y}$ es el espacio de salida (conjunto de las posibles salidas). Existe entonces un conjunto $\mathbf{D}$ de ejemplos entrada-salida $(x_1, y_1), ..., (x_n, y_n)$, donde $y_n = f(x_n)$ para $n = 1, ..., N$. Finalmente hay un algoritmos de aprendizaje que utiliza el conjunto $\mathbf{D}$ para buscar una fórmula $g : X \rightarrow Y$ que aproxime f. El algoritmo elige g de un conjunto de fórmulas candidatas, al cual se llama conjunto de hipótesis H [1].

En general cualquier problema de aprendizaje de máquina es clasificado en:

- Conocimiento supervisado

- Conocimiento no supervisado

2.2. Tipos de conocimiento

Uno de los principios básicos del aprendizaje de máquina es aprender a partir de un conjunto de datos. Debido a la naturaleza de dichos datos es que se han definido distintos tipos de conocimiento, entre los que destacan el aprendizaje supervisado y el aprendizaje no supervisado; aunque no son los únicos.

2.2.1. Conocimiento supervisado

Cuando se cuenta con un conjunto de datos, del cual se conoce la salida correcta después de aplicarle un algoritmo de aprendizaje de máquina, se dice que se está frente a un problema de conocimiento supervisado, pues se tiene una idea de la relación que existe entre la entrada y la salida del algoritmo necesario para resolver dicho problema.

El conocimiento supervisado es comúnmente divido en dos categorías: regresión y clasificación. En un problema de regresión se trata de predecir resultados con una salida continua, es decir se intenta asignar variables de entrada a una función continua. Por otro lado en un problema de clasificación lo que se busca es predecir resultados con una salida discreta; en otras palabras lo que se busca es asignar las variables de entrada en categorías discretas.

2.2.2. Conocimiento no supervisado

Por otro lado, el conocimiento no supervisado se presenta cuando se desconoce parcialmente o por completo la salida correcta que se obtendría al aplicarle un algoritmo de aprendizaje al conjunto de datos con el que se cuenta. El objetivo del aprendizaje no supervisado es encontrar la estructura que mejor represente a los datos de entrada [35]. Los métodos del conocimiento no supervisado se realizan sin ningún tipo de guía por lo que la estructura que se obtiene con ellos de los datos, no siempre es relevante para el análisis de los datos.

2.3. Agrupamiento (Clustering)

Agrupamiento es considerado como el problema más representativo del conocimiento no supervisado; cuyo objetivo es dividir un conjunto de datos no clasificados en subconjuntos a los cuales se les nombra clúster y hacer válidas las siguientes preposiciones [46]:

1. Los datos que son colocados en el mismo clúster, deben ser lo más similares entre sí.
2. Los datos que sean colocados en clústeres distintos, deben ser entre sí, lo más diferentes posible.

Los algoritmos de agrupamiento varían entre sí por las reglas que utilizan (heurísticas y/o probabilistas) y el tipo de aplicación para el cual fueron diseñados. La mayoría se basa en empleo sistemático de distancias entre vectores, así como entre clústeres o grupos que se van formando a lo largo del proceso de agrupamiento [2].

2.3.1. Algoritmos de agrupamiento tradicionales

A continuación se muestra una clasificación para algoritmos tradicionales de agrupamiento con base en los más comunes dentro del estado del arte (ver tabla 2.1)

Algoritmos de agrupamiento basados en partición

Estos algoritmos producen un número de clústeres que es definido desde el inicio, donde cada punto pertenece a un clúster en particular. Un clúster es representado por un centro de clúster o centroide el cual es una descripción de todos los puntos que pertenecen a ese clúster en particular. La definición de centroide depende del tipo de datos que estemos agrupando, es decir, el centroide puede estar dado por una media aritmética, o por la medida modal de un conjunto de palabras. Algunos ejemplo son el algoritmo K-medias, K-medoids, PAM (Partitioning Around Medoids), CLARA (Clustering LARge Applications), CLARANS (CLAra based on RANdomized Search). [2]

Algoritmos de agrupamiento basados en jerarquías

Los algoritmos de agrupamiento basados en jerarquías tienen por objetivo agrupar clústeres para formar uno nuevo o bien separar algún clúster que ya exista para dar origen a otros dos, de tal forma que, si sucesivamente se va efectuando este proceso de

Categoría	Algoritmo
Algoritmos de agrupamiento basados en partición.	K-medias, K-medoids, PAM, CLARA, CLARANS.
Algoritmos de agrupamiento basados en jerarquías.	BIRCH, CURE, ROCK, Chameleon.
Algoritmos de agrupamiento basados en lógica difusa.	FCM, FCS, MM.
Algoritmos de agrupamiento basados en distribución.	DBCLASD, GMM.
Algoritmos de agrupamiento basados en densidad.	DBSCAN, OPTICS, Mean-shift.
Algoritmos de agrupamiento basados en teoría de grafos.	CLICK, MST.
Algoritmos de agrupamiento basados en grid.	STING, CLIQUE.
Algoritmos de agrupamiento basados en teoría de fractales.	FC.
Algoritmos de agrupamiento basados en modelo.	COBWEB, GMM, SOM, ART.

Tabla 2.1: Algoritmos tradicionales

aglomeración o división, se minimice alguna distancia o bien se maximice alguna medida de similitud. A su vez los algoritmos jerárquicos pueden ser clasificados en dos tipos [2]:

- Aglomerativos. Son un enfoque *bottom-up* donde cada punto empieza en su propio clúster, y pares de clústeres son unidos a medida que se avanza en la jerarquía.
- Divisivos. Son un enfoque *top-down*, comienza en el supuesto de que todos los datos están en un solo clúster y se realizan divisiones de forma recursiva a medida en que se desciende en la jerarquía.

Ejemplos de este tipo de algoritmos son, BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies), CURE (Clustering Using REpresentatives), ROCK (RObust Clustering using linKs), Chameleon.

Algoritmos de agrupamiento basados en lógica difusa

La idea básica de este tipo de algoritmos es que el valor discreto de la etiqueta de pertenencia {0,1}, se cambie por un valor en el intervalo [0,1], con el fin de describir la

relación de pertenencia entre los objetos más razonable. Entre las ventajas que presentan podemos encontrar que este tipo de algoritmos de agrupamiento son más realistas para dar la probabilidad de pertenencia para cada dato, lo cual relativamente da una alta precisión de agrupación. Por otro lado, es muy propenso a caer en puntos óptimos locales. Algunos ejemplos son: FCM (Fuzzy c-Medias Clustering), FCS.

Algoritmos de agrupamiento basados en distribución

Este tipo de algoritmos se basa en la idea de que los datos generados a partir de una misma distribución de probabilidad pertenecen a un mismo clúster si existen varias distribuciones de los datos. Ejemplos de este tipo de algoritmos son DBCLASD (Distribution-Based Clustering of LArge Spatial Databases) y GMM (Gaussian Mixture Models) mismo que se implementa en el presente trabajo de tesis y del cual hablaremos más a detalle en los siguientes apartados.

Algoritmos de agrupamiento basados en densidad

La idea general de estos algoritmos se basa en localizar las zonas de alta densidad separadas por regiones de baja densidad. El algoritmo más representativo es DBSCAN (Density-based spatial clustering of applications with noise).

Algoritmos de agrupamiento basados en teoría de grafos

En este tipo de algoritmos de agrupación, el proceso de crear clústeres se realiza en el grafo donde el nodo es considerado como el punto de datos y la arista se considera como la relación entre los puntos dados. Los algoritmos que se identifican dentro de esta categoría son: CLICK (CLuster Identification via Connectivity Kernels) y MST-based clustering.

Algoritmos de agrupamiento basados en *grid*

La idea básica de este tipo de algoritmos de agrupamiento es que el espacio de datos original se transforma en una estructura de rejilla con un tamaño definido para el agrupamiento. Ejemplos de este tipo de algoritmos son: STING (STatistical INformation

Grid-based method) y CLIQUE (CLustering In QUEst).

Algoritmos de agrupamiento basados en teoría de fractales

Los fractales representan geométricamente que el conjunto de datos puede ser dividido en varias partes que comparten características con el resto de los elementos del conjunto. El algoritmo típico es el FC del cual la idea central es que el cambio de los datos internos de un clúster no tiene ninguna influecia en la calidad intrínseca de la dimensión fractal.

Algoritmos de agrupamiento basados en modelo

La idea básica es seleccionar un modelo en particular para cada clúster y encontrar la mejor ajuste para este modelos. Dentro de esta categoría podemos encontrar dos categorías más, una se basa en el método de aprendizaje estadístico y otra basada en el aprendizaje de una red neuronal. Los algoritmos típicos basados en el método de aprendizaje estadístico son COBWEB y GMM (Gaussians Mixtures Model). Mientras que para los basados en el aprendizaje de una red neuronal son SOM (Self-Organizing Map) y ART.

2.3.2. Distancia y similitud

Distancia (disimilitud) y similitud son conceptos básicos para la construcción de algoritmos de agrupamiento, de acuerdo al tipo de datos que se manejen como entrada. La similitud cuantifica principalmente la relación entre las características de diferentes objetos. Las medidas de distancia son las más utilizadas para cuantificar la similitud entre dos objetos, es por ello que la elección de la función distancia es fundamental para cualquier algoritmo de agrupamiento.

Suponga que X denota un conjunto de n puntos $X = \{\bar{x_1}, \bar{x_2}, ..., \bar{x_n}\}$. Sea d la dimensión de los datos. Entonces $x_{ij}, i = 1, ..., n$ y $j = 1, ..., d$, denota el valor de la j-ésima dimensión del i-ésimo punto. La distancia entre dos datos $\bar{x_i}$ y $\bar{x_j}$ denotada por $D(\bar{x_i}, \bar{x_j})$. Entonces esta función de distancia debe satisfacer las siguientes propiedades:

1. $D(\bar{x_i}, \bar{x_j}) \geq 0 \ \forall \ \bar{x_i}, \bar{x_j} \in \boldsymbol{X}$.
2. $D(\bar{x_i}, \bar{x_j}) = 0$ si y solo si $i = j$.

3. $D(\bar{x}_i, \bar{x}_j) = D(\bar{x}_j, \bar{x}_i)\ \forall\ \bar{x}_i, \bar{x}_j \in \boldsymbol{X}$.

4. $D(\bar{x}_i, \bar{x}_j) \leq D(\bar{x}_i, \bar{x}_l) + D(\bar{x}_l, \bar{x}_j)\ \forall\ \bar{x}_i, \bar{x}_j, \bar{x}_l \in \boldsymbol{X}$.

Cuando una función de distancia satisface las condiciones anteriores se le denomina métrica [2]. En la literatura se han empleado bastantes medidas para desarrollar algoritmos de agrupamiento, la más común es la distancia euclidiana (la distancia euclidiana más pequeña significa la mayor similitud, y viceversa); esta medida se utiliza en un algoritmo clásico de agrupamiento K-Medias aunque no es la única (ver tabla 2.2 [46]).

Nombre	Fórmula	Descripción
Minkowski	$(\sum_{l=1}^{d} \lvert x_{il} - x_{jl}\rvert^n)^{1/n}$	1. Distancia Manhattan cuando $n = 1$. 2. Distancia Euclidiana cuando $n = 2$. 3. Distancia Chebyshev cuando $n \to \infty$.
Euclidiana	$(\sum_{l=1}^{d} \lvert x_{il} - x_{jl}\rvert^2)^{1/2}$	1. Es la distancia más utilizada.
Coseno	$1 - \cos\alpha = \frac{x_i^T x_j}{\lvert\lvert x_i\rvert\rvert\lvert\lvert x_j\rvert\rvert}$	1. La similaridad coseno no debe ser considerada como una métrica debido a que no cumple la desigualdad triangular 2. Suele emplearse como un indicador de cohesión de clústeres de textos.
Pearson	$1 - \frac{Cov(x_i, x_j)}{\sqrt{D(x_i)}\sqrt{D(x_j)}}$	1. Se basa en la correlación lineal. 2. Cov = covarianza y D = varianza.
Mahalanobis	$\sqrt{(x_i - x_j)^T S^{-1} (x_i - x_j)}$	1. S representa a la matriz de covarianza. 2. Es de gran complejidad computacional.

Tabla 2.2: Funciones de distancia

Cada medida tiene ventajas y desventajas que las hacen más o menos apropiadas dependiendo de la naturaleza de los datos de entrada.

2.4. K-Medias (K-Means)

Agrupamiento cuenta con una rica y diversa historia en distintos campos de la ciencia. Uno de los más populares y simples algoritmos de agrupamiento por partición es K-medias, el cual fue publicado por primera vez en 1955. A pesar de que K-medias fue propuesto hace más de 60 años y de que miles de algoritmos de agrupamiento han sido propuestos desde entonces, K-medias es todavía muy utilizado cuando de problemas de agrupamiento se trata [**jain1999data**].

El algoritmo K-medias es una técnica iterativa de agrupamiento que involucra K compactos e hiperesféricos clústeres, entendiendo por hiperesférico a un conjunto de puntos con una distancia constante dada por un punto al cual se le denomina centro.

Dado un conjunto de datos se agrupan en K número de clústeres disjuntos, donde el valor de K es definido previamente. Dado $X = \{x_i\}\ i = 1, ..., n$ como el conjunto de n-dimensional puntos que serán agrupados y $C = \{c_k, \}\ k = 1, ..., K$ los clústeres que resultarán; K-medias encuentra una partición tal que el error cuadrado entre el centroide de un clúster y cada uno de los puntos que se encuentren dentro de ese clúster será mínimo. Dado μ_k como el centroide del clúster c_k. El error cuadrado entre μ_k y el punto i en el clúster c_k esta definido por:

$$J(c_k) = \sum_{x_i \in c_k} ||x_i - \mu_k||^2 \tag{2.1}$$

El objetivo es minimizar una función objetivo, la cual suele ser llamada medida de distorsión:

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} ||x_n - \mu_k||^2 \tag{2.2}$$

Donde r_{nk} representa por cada punto x_n un esquema de codificación "1-of-k", es decir por cada x_n se introduce un conjunto de variables indicadoras binarias $r_{nk} \in \{0, 1\}$, donde k = 1,...,K describe a cuál de los K clústeres ha sido asignado x_n, tal que si x_n ha sido asignado al clúster k, entonces $r_{nk} = 1$ y $r_{nj} = 0$ para $j \neq k$ [3]. Se pretende encontrar valores para r_nk y los centroides μ_k tal que J sea mínima.

Podemos lograr esto a través de un proceso iterativo, en el cual cada iteración involucra dos pasos correspondientes a dos sucesivas optimizaciones con respecto a r_{nk} y μ_k. Primero elegimos valores aleatorios para μ_k, entonces en la primera fase minimizamos J con respecto a r_{nk} manteniendo el μ_k fijado anteriormente. En la segunda fase minimizamos J con respecto a μ_k manteniendo el r_{nk} que ha sido fijado. Ambos pasos de optimización deben repetirse hasta que se logre la convergencia. La medida de la distancia de cada punto a los respectivos centroides puede ser calculada de diferentes maneras, utilizando la distancia Mahalanobis, de Minkowski, euclidiana, entre otras. Para este trabajo se utilizó la distancia euclidiana.

Algorithm 1 Algoritmo K-medias

Require: Conjunto de **N** datos, **K**, condición de paro 1 **STOP1**, y condición de paro 2 **STOP2**.

GPU Obtener los **K** centroides aleatorios; $z_1, z_2, ..., z_K$

Paso 1: Elegir K centroides aleatorios.

Paso 2: Asignar el punto x_i, $i = 1, 2, ..., K$ al clúster $C_j \in 1, 2, ..., K$ si y sólo si $||x_i - z_j|| < ||x_i - z_p||$, $p = 1, 2, ..., K$, y $j \neq p$ los empates se resuelven arbitrariamente.

Paso 3: Calcular los nuevos centroides $z_1^*, z_2^*, ..., z_k^*$ de la siguiente forma:
$z_i^* = \frac{\sum_{x_j \in C_i} x_j}{n_i}$, $i = 1, 2, ..., K$
donde n_i es el número de elementos que pertenecen al clúster C_i

Paso 4: Si $z_i^* = z_i, i = 1, 2, ..., K$ entonces termina; de otra forma, $z_i = z_i^*, i = 1, 2, ..., K$ y continua desde el paso 2.

2.5. Mezcla de Gausianas

Este algoritmo se basa en el uso de una de las más importantes distribuciones de probabilidad para variables continuas: la distribución Gaussiana o distribución Normal. Sólo se dará un pequeño repaso a los conceptos más relevantes de esta distribución.

La distribución Gaussiana depende de dos parámetros: la *media*, denotada generalmente por μ y la *varianza* denotada por σ^2; la raíz cuadrada de la varianza está dada por la desviación estándar σ. El recíproco de la varianza, escrito como $\beta = 1/\sigma^2$, es llamado *precisión* (ver figura 2.2).

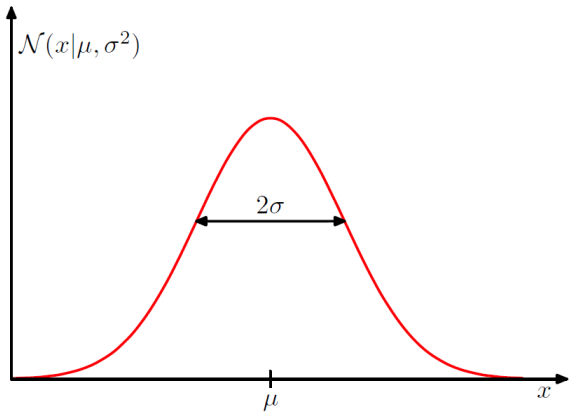


Figura 2.2: Representación gráfica de la distribución Gaussiana univariante.

Para el caso de una variable real x, la distribución Gaussiana esta definida por:

$$\mathcal{N}(x|\mu,\sigma^2) = \frac{1}{(2\pi\sigma^2)^{1/2}} exp\{-\frac{1}{2\sigma^2}(x-\mu)^2\} \tag{2.3}$$

Para la ecuación 2.3 podemos ver que la distribución Gaussiana satisface lo siguiente:

$$\mathcal{N}(x|\mu,\sigma^2) > 0. \tag{2.4}$$

También es sencillo demostrar que se puede normalizar quedando:

$$\int_{-\infty}^{\infty} \mathcal{N}(x|\mu,\sigma^2)dx = 1. \tag{2.5}$$

por lo tanto 2.3 satisface los dos requisitos para una densidad de probabilidad válida.

Podemos encontrar fácilmente la esperanza en función de x dada por:

$$\mathbb{E}[x] = \int_{-\infty}^{\infty} \mathcal{N}(x|\mu, \sigma^2) x dx = \mu. \tag{2.6}$$

El parámetro μ representa el valor promedio de x bajo la distribución, por lo cual es llamado media. Calculando el segundo momento:

$$\mathbb{E}[x^2] = \int_{-\infty}^{\infty} \mathcal{N}(x|\mu, \sigma^2) x^2 dx = \mu^2 + \sigma^2. \tag{2.7}$$

De 2.6 y 2.7 se deduce que la varianza de x está dada por:

$$var[x] = \mathbb{E}[x^2] - \mathbb{E}[x]^2 = \sigma^2. \tag{2.8}$$

por lo tanto se conoce a σ como el parámetro de varianza.

Para el presente trabajo es de gran interés la definición de la distribución Gaussiana sobre un vector x D-dimensional de variables continuas, la cual está dada por:

$$\mathcal{N}(x|\mu, \Sigma) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\Sigma|^{1/2}} exp\{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)\} \tag{2.9}$$

donde el vector D-dimensional es llamado *media* y la matriz (DxD) Σ es la matriz de covarianza, y $|\Sigma|$ denota al determinante de Σ.

Aunque la distribución Gaussiana tiene algunas propiedades analíticas importantes, sufre de limitaciones significativas cuando se trata de modelado de conjuntos de datos reales (los datos que son la típica entrada para resolver cualquier problema de aprendizaje de máquina). En específico al trabajar con algoritmos de agrupamiento donde se desea encontrar una estructura para los datos donde exista más de un grupo dominante, como lo haría una distribución Gaussiana, es posible que una superposición lineal de dos o k distribuciones Gaussianas proporcione una mejor caracterización del conjunto de datos con el que se trabaja.

Tales superposiones, formadas mediante combinaciones lineales de las distribuciones más básicas (como la Gaussiana) pueden formularse como modelos probabilistas conocidos como *Mezcla de distribuciones* (mixture distributions) [31] [30]. En la figura 2.3 es posible apreciar como una combinación lineal de Gaussianas puede dar lugar a densidades muy

complejas. Mediante el uso de un número suficiente de Gaussianas y mediante el ajuste de sus medias y covarianzas así como de los coeficientes en la combinación lineal, casi cualquier densidad continua puede ser aproximada.

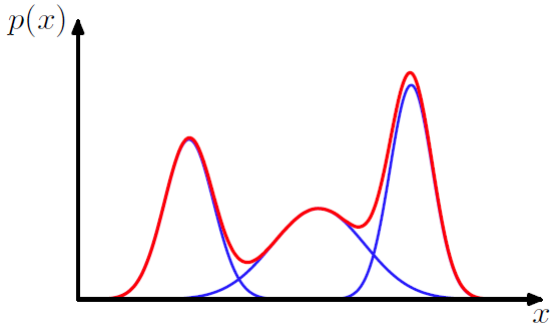


Figura 2.3: Ejemplo de una distribución de mezcla Gaussiana en una dimensión mostrando 3 Gaussianas en azul y su suma en color rojo.

Es entonces que podemos considerar una superposición de K Gaussianas como:

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k.\Sigma_k). \tag{2.10}$$

la cual recibe el nombre de *Mezcla de Gaussianas*. Cada densidad Gaussiana $\mathcal{N}(x|\mu_k, \Sigma_k)$ es llamada *componente* de la mezcla y tiene su propia media μ_k y su propia matriz de covarianza Σ_k.

Los parámetros π_k en 2.10 son llamados coeficientes de mezcla. Si realizamos la integración en 2.10 con respecto a x, y observamos que tanto $p(x)$ y los componentes Gaussianos son normalizados obtenemos:

$$\sum_{k=1}^{K} \pi_k = 1. \tag{2.11}$$

Además el requerimiento de que $p(x) \geq 0$, junto con $\mathcal{N}(x|\mu_k, \Sigma_k) \geq 0$, implica que $\pi_k \geq 0$ para toda k. Combinando esto con la condición 2.11 tenemos:

$$0 \leq \pi_k \leq 1. \tag{2.12}$$

Por lo tanto, los coeficientes de mezcla satisfacen el requisito principal para ser probabilidades. A partir de las reglas de suma y producto, la densidad marginal está dada

por:

$$p(x) = \sum_{k=1}^{K} p(k)p(x|k) \tag{2.13}$$

ecuación que es equivalente a 2.10 en que podemos ver a $\pi_k = p(k)$ como la probabilidad a priori del k-ésimo componente y a la densidad $\mathcal{N}(x|\mu_k, \Sigma_k) = p(x|k)$ como la probabilidad de x condicionada a k; así pues la probabilidad a posteriori juega un importante rol $p(k|x)$ el cual es conocido como responsabilidades (responsibilities). Haciendo uso del teorema de Bayes, estas responsabilidades están dadas por:

$$\begin{aligned} \gamma_k(x) &\equiv p(k|x) \\ &= \frac{p(k)p(x|k)}{\sum_l p(l)p(x|l)} \\ &= \frac{\pi_k \mathcal{N}(x|\mu_k, \Sigma_k)}{\sum_l \pi_l \mathcal{N}(x|\mu_l, \Sigma_l} \end{aligned} \tag{2.14}$$

Como ya se mencionó, la distribución de mezcla de gaussianas está dada por los parámetros π, μ y Σ, donde $\pi \equiv \{\pi_1, ..., \pi_K\}$, $\mu \equiv \{\mu_1, ..., \mu_K\}$, $\Sigma \equiv \{\Sigma_1, ..., \Sigma_K\}$. Una forma de darles valores a estos parámetros es utilizando *máxima verosimilitud*, a través de maximizar la probabilidad utilizando la función logaritmo, que se comporta de una forma creciente, y está dado por:

$$ln(p(X|\pi, \mu, \Sigma)) = \sum_{n=1}^{N} ln\{\sum_{k=1}^{K} \pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)\} \tag{2.15}$$

donde $X = \{x_1, ..., x_N\}$. [3]

2.6. Algoritmo EM

El algoritmo EM (Expectation-Maximization) es una técnica general para encontrar la máxima verosimilitud para modelos probabilísticos con variables latentes.

Este algoritmo procede en dos pasos que se repiten de manera iterativa:

- **Paso E (Expectation):** Utiliza los valores de los parámetros, iniciales o proporcionados por el paso *Paso M (Maximization):* de la iteración anterior, para obtener diferente formas de la función de densidad de probabilidad.

- **Paso M (Maximization):** Obtiene nuevos valores de los parámetros a partir de los datos proporcionados por el paso anterior.

Bishop [3] presenta el desarrollo analítico de este algoritmo:

Considerando cualquier modelo probabilístico donde X denota a todas las variables observadas y Z a las variables latentes. La distribución conjunta $p(X, Z|\Theta)$ está dada por los parámetros denotados por Θ. El objetivo es maximizar la función de probabilidad dada por:

$$p(X|\Theta) = \sum_z p(X, Z|\Theta). \tag{2.16}$$

Asumimos que Z es discreta, aunque no hay diferencia si se asume que Z incluye variables continuas o es una combinación de ambas (discretas y continuas) utilizando una integral en lugar de la sumatoria.

Suponemos que el problema de optimización directo $p(Z|\Theta)$ es sumamente difícil, pero optimizar la función $p(X, Z|\Theta)$ es significativamente más sencilla. Posteriormente se introduce una distribución $q(Z)$ definida sobre las variables latentes, y es posible observar que para cualquier elección de $q(Z)$, es válida la siguiente descomposición:

$$ln(p(X|\Theta)) = \mathcal{L}(q, \Theta) + KL(q||p) \tag{2.17}$$

Donde se definen

$$\mathcal{L}(q, \Theta) = \sum_z q(Z) ln\{\frac{p(X, Z|\Theta)}{q(Z)}\} \tag{2.18}$$

$$KL(q||p) = -\sum_z q(Z) ln\{\frac{p(Z|X, \Theta)}{q(Z)}\} \tag{2.19}$$

$\mathcal{L}(q, \Theta)$ es un funcional de la distribución $q(Z)$ y una función de parámetros Θ. Es resulta importante notar que las expresiones 2.17 y 2.18 difieren en signo y que 2.17 contiene la distribución adjunta de X y Z, mientras que 2.18 contiene la distribución condicional de Z dado X. Para verificar 2.16 se hace uso de la regla del producto:

$$ln(p(X, Z|\Theta)) = ln(p(Z|X, \Theta)) + ln(p(X|\Theta)) \tag{2.20}$$

La cual sustituimos en la expresión $\mathcal{L}(q, \Theta)$. Esto da lugar a dos términos, uno de los cuales cancela $KL(q||p)$ mientras que el otro da el logaritmo de la probabilidad $ln(p(X|\Theta))$ después de notar que $q(Z)$ es una distribución normalizada que suma 1. De 2.18 se puede notar que $KL(q||p)$ es la divergencia de Kullback-Leibler entre $q(Z)$ y la distribución posteori $p(Z|X, \Theta)$. La divergencia de Kullback-Leibler satisface $KL(q||p) \geq 0$ con igualdad si y solo si $q(Z) = p(Z|X, \Theta)$. Por lo tanto de 2.17 resulta que $\mathcal{L}(q, \Theta) \leq lnp(X|\Theta)$ en otras palabras $\mathcal{L}(q, \Theta)$ es una cota inferior de $lnp(X|\Theta)$.

Sólo como definición. Se puede pensar en una función $y(x)$ como un operador que, para cualquier valor de x devuelve un valor salida y. De la misma forma se puede definir un funcional $F[y]$ para ser un operador que toma una función $y(x)$ y devuelve un valor de salida F. Un ejemplo de un funcional es la longitud de una curva dibujada en un plano bidimensional, en el cuál la trayectoria de la curva se define en términos de un función. En el contexto de aprendizaje maquina un funcional ampliamente utilizado es la entropía $H[x]$ para una variable continua x porque para cualquier función de densidad de probabilidad $p(x)$ esta regresa un valor escalar que representa la entropía bajo esa densidad.

Un problema común en el cálculo convencional es encontrar un valor de x que maximiza (o minimiza) una función $y(x)$. Del mismo modo, en el cálculo de variaciones se busca una función $y(x)$ que maximice o minimice un funcional $F[y]$. Es decir, para todas las posibles funciones $y(x)$, se desea encontrar una función particular para la cuál el funcional $F[y]$ es máximo (o mínimo). El cálculo de variaciones puede ser utilizado para mostrar que la ruta más corta entre dos puntos es una línea recta o que la distribución de máxima entropía es una gaussiana.

Como ya se mencionó, el Algoritmo EM es una técnica iterativa y de dos paso, para encontrar soluciones de máxima probabilidad. Es posible utilizar la ecuación 2.17 para definir el algoritmo EM y demostrar que efectivamente maximiza la función logaritmo de la probabilidad. Suponiendo que el valor actual del vector de parámetros es Θ^{old}. En el paso E la cota inferior $\mathcal{L}(q, \Theta^{old})$ es maximizada respecto a $q(Z)$ mientras Θ^{old} se mantiene fijo. La solución a este problema de optimización se facilita debido a que el valor de $lnp(X|\Theta^{old})$ no depende de $q(Z)$ y por lo tanto el valor máximo de $\mathcal{L}(q, \Theta^{old})$ ocurre cuando $q(Z)$ es igual a la distribución posteori $p(Z|X, \Theta^{old})$; en este caso, la cota inferior será igual a logaritmo de la probabilidad.

En el siguiente paso M, la distribución $q(Z)$ se mantiene fijo y el límite inferior $\mathcal{L}(q, \Theta)$ se maximiza con respecto a Θ dando como resultado un nuevo valor Θ^{new}. Esto puede ocasionar que la cota inferior $\mathcal{L}$ incremente (a menos que ya este en un máximo), lo que necesariamente hará que la función logaritmo de la probabilidad correspondiente aumente. Debido a que la distribución q se determina utilizando los parámetros antiguos en lugar de los nuevos y se mantiene fijos durante el paso M, no será igual a la nueva distribución posteori $p(Z|X, \Theta^{new})$, y por lo tanto existirá una divergencia KL distinta de 0. El incremento en la función logaritmo de la probabilidad es mayor que el incremento en el límite inferior. Si se sustituye $q(Z) = p(Z|X, \Theta^{old})$ en 2.18 se observa que después del paso E la cota inferior toma la forma:

$$\begin{aligned} \mathcal{L}(q, \Theta) &= \sum_z p(Z|X, \Theta^{old}) lnp(X, Z|\Theta) - \sum_z p(Z|X, \Theta^{old}) lnp(Z|X, \Theta^{old}) \\ &= \mathcal{Q}(\Theta, \Theta^{old}) + const \end{aligned} \tag{2.21}$$

Donde la constante es simplemente la entropía negativa de la distribución q y por lo tanto, es independiente de Θ. Por lo tanto, en el paso M, la cantidad que se maximiza es la esperanza del logaritmo de la probabilidad de los datos completos. La variable Θ sobre la que se optimiza aparece dentro del logaritmo. Si la distribución conjunta $p(Z, X|\Theta)$ comprende a un miembro de la familia exponencial, o un producto de tales miembros, entonces el logaritmo cancelará la exponencial y dará un paso M mucho más simple que la maximización de la correspondiente función logaritmo de la probabilidad de los datos incompletos $p(X|\Theta)$.

Aunque EM garantiza convergencia, ésta puede representar un máximo local.

2.7. Programación paralela

Debido a los grandes cambios en cuanto a la forma de hacer y organizar la información de hoy en día, la idea de un solo procesador se está volviendo rápidamente arcaica, ahora se tienen que ajustar las estrategias de computación para dar solución a los problemas que dichos cambios representan:

- Es imposible mejorar el rendimiento de una computadora mediante un único procesador. Tal procesador consumiría demasiada energía. Es más práctico utilizar muchos procesadores simples para alcanzar el rendimiento deseado utilizando quizá miles de tales ordenadores simples [44].

- Se tiene que desarrollar herramientas que sean capaces de detectar el paralelismo de un algoritmo dado. Un algoritmo puede mostrar la dependencia habitual entre sus variables, o que la dependencia podría ser irregular. En cualquier caso, hay espacio para la aceleración de la ejecución del algoritmo siempre que algunas subtareas puedan ejecutarse simultáneamente mientras se asegure la exactitud de la ejecución.

- Los beneficios del cómputo paralelo deben tener en cuenta el número de procesadores sobre los que se están desplegado así como la sobrecarga de comunicación de procesador a procesador y del procesador a la memoria. Los problemas de cómputo pueden presentarse cuando el aumento de la velocidad depende de la velocidad de ejecución del algoritmo por el procesador. Los problemas de comunicación son aquellos en los que el aumento potencial de velocidad depende de la velocidad de suministro de los datos a y de la extracción de los datos de los procesadores.

- Los sistemas de memoria son todavía más lentos que los procesadores y su ancho de banda está limitado, en una palabra por ciclo lectura/escritura.
- Los científicos, y programadores ya no se adaptan a las necesidades de cómputo para desarrollar sus algoritmos; en lugar de esto están tratando de adaptar el hardware para solucionar sus requisitos de cómputo.

Como se ha mencionado los problemas del aprendizaje de máquina son influenciados por la enorme cantidad información con la que se cuenta y a la que se quiere dar un procesamiento para así poder resolver los problemas de conocimiento que se presentan. Dentro del presente trabajo de tesis se ha decidido utilizar dos plataformas de cómputo paralelo, mismas de las que se da una breve descripción a continuación.

2.7.1. CUDA

CUDA es una arquitectura de cálculo paralelo de NVIDIA Corporation que aprovecha la gran potencia de la GPU (unidad de procesamiento gráfico) para proporcionar un incre-

mento extraordinario del rendimiento del sistema. Permite al usuario utilizar una variación del lenguaje de programación C para codificar algoritmos. Por medio de *wrappers*(APIs que permiten el uso de las sentencias de un lenguaje de programación, dentro de otro lenguaje de programación) se puede usar Python, Fortran y Java en vez de C/C++ y en el futuro también se añadirá FORTRAN, OpenGL y Direct3D.

La arquitectura de CUDA es descrita como un multiprocesador contiene ocho procesadores escalares, dos unidades especiales para funciones trascendentales, una unidad multihilo de instrucciones y una memoria compartida. El multiprocesador crea y maneja los hilos sin ningún tipo de *overhead* (tiempo desperdiciado por el procesador para realizar un cambio de contexto) por la planificación, lo cual unido a una rápida sincronización por barreras y una creación de hilos muy ligera, consigue que se pueda utilizar CUDA en problemas de muy baja granularidad, incluso asignando un hilo a un elemento por ejemplo de una imagen (un píxel).

CUDA intenta explotar las ventajas de las GPU frente a las CPU de propósito general utilizando el paralelismo que ofrecen sus múltiples núcleos, que permiten el lanzamiento de un altísimo número de hilos simultáneos. Por ello, si una aplicación está diseñada utilizando numerosos hilos que realizan tareas independientes (como lo hacen las GPU al procesar gráficos, su tarea natural), una GPU podrá ofrecer un gran rendimiento en campos que podrían ir desde la biología computacional a la criptografía, por ejemplo.

CUDA define principalmente tres cosas:

1. Un modelo de arquitectura.
2. Un modelo de programación.
3. Un modelo de gestión de la memoria.

Para codificar algoritmos en GPU existen un conjunto de herramientas desarrolladas por nVidia que permiten utilizar una variación del lenguaje de programación C; por medio de wrappers (herramientas que permiten transformar una interfaz en otra) es posible también utilizar lenguajes como Python, Fortran, C#, entre otras. Para fines del presente trabajo, se utilizará el lenguaje C (las razones se expondrán más adelante).

A continuación se presenta un pequeño repaso de algunos conceptos de CUDA que resultan importantes para el presente trabajo.

Arquitectura de los GPUs

Una GPU está compuesta por N multiprocesadores. A su vez, cada uno está compuesto por M núcleos (cores). Se puede tener un paralelismo masivo aplicando sobre miles de hilos que comparten datos a diferentes niveles de una jerarquía de memoria. Es entonces como el GPU puede tener el 5 % del código fuente del programa lo que puede reducir hasta en un 50 % el tiempo de ejecución del mismo.

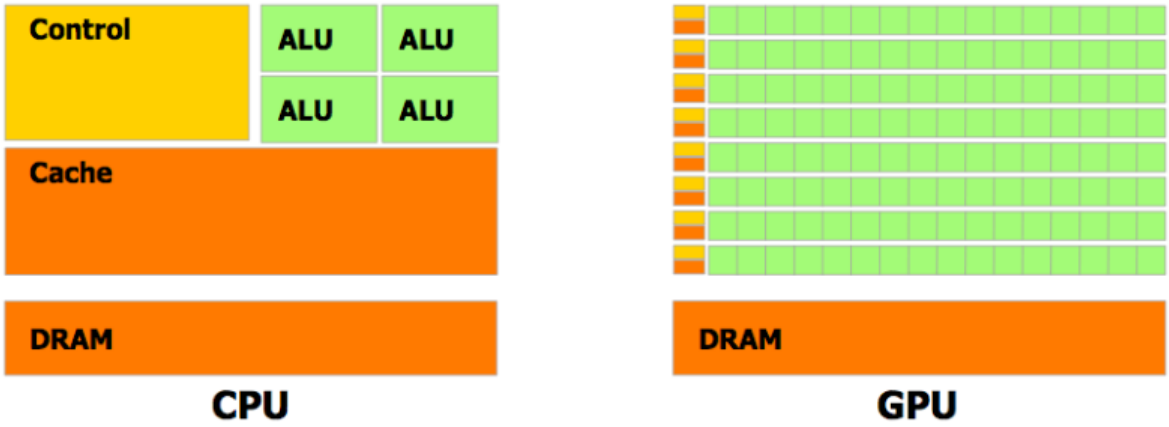


Figura 2.4: CPU vs GPU.

Como se muestra en 2.4, este modelo requiere de menos lógica de control para cada flujo de ejecución. Al mismo tiempo, se dispone de una pequeña memoria caché que permite que flujos que comparten memoria tengan el ancho de banda suficiente para no tener que ir todos a la DRAM, provocando con esto que más área del GPU se dedique al procesamiento de datos. Las arquitecturas GPU siguen el modelo SIMD (Single Instruction, Multiple Data) que es una técnica en la que todos los núcleos (*cores*) ejecutan a la vez una misma instrucción. La memoria de una GPU se organiza en varios tipos de memoria (local, global, constante y textura), que tienen diferentes tiempos y modos de acceso (solo lectura o lectura/escritura) así como distintos tamaños.

Jerarquía de memoria

Cada multiprocesador cuenta con su propio:

- Banco de registros.
- Memoria compartida.

- Caché de constantes y otra de texturas, ambas de solo lectura y uso marginal.

Además existe una memora global que es la memoria de video, la cual es tres veces más rápida que la memoria principal de la CPU, pero quinientas veces más lenta que la memoria compartida.

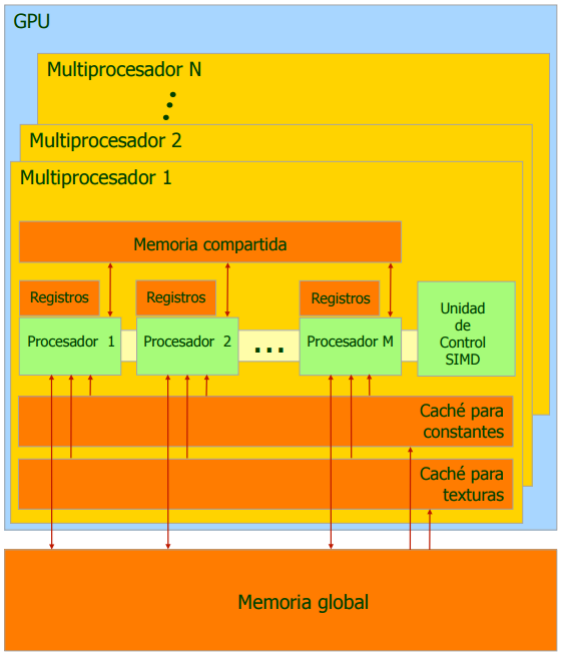


Figura 2.5: Jerarquía memoria.

Flujo de procesamiento.

Dentro de los objetivos de CUDA se encuentra el permitir la computación heterogénea en *Host* y *Device* (CPU y GPU, respectivamente). CUDA ejecuta una secuencia de código sobre un dispositivo *device*, que actúa como coprocesador de un anfitrión el que recibe el nombre de *host*.

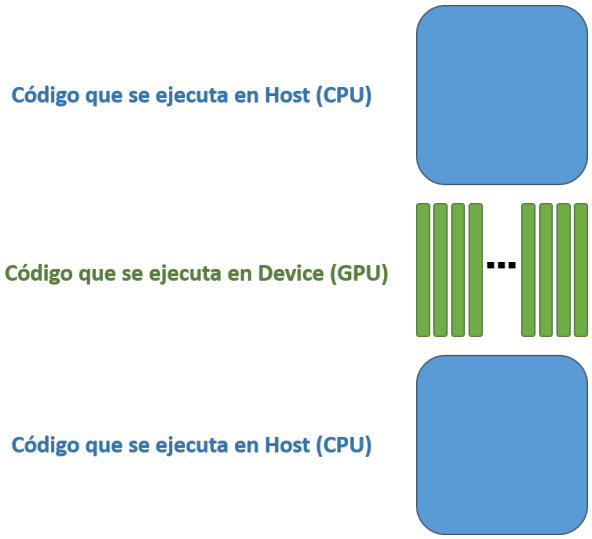


Figura 2.6: Representación gráfica del flujo de un programa en CUDA.

Tanto el *host* como el *device* poseen espacios de memoria (DRAM) separados, las cuales pueden tener comunicación entre sí utilizando el bus PCI-express. Es por ello que un programa escrito para trabajar con GPUs sigue el flujo que se describe a continuación:

- Copiar los datos de entrada de la memoria de la CPU a la memoria de la GPU.
- Cargar el programa en la GPU y ejecutarlo, ubicando datos en caché para mejorar el rendimiento.
- Transferir los resultados de la memoria de la GPU a la memoria de la CPU.

CUDA como modelo de programación.

El objetivo de un programador que decide trabajar con CUDA es: *Declarar miles de hilos, que la GPU necesita para lograr rendimiento y escalabilidad.* Los hilos de CUDA son extremadamente ligeros, se crean en un tiempo muy efímero y la conmutación de contexto

es inmediata.

Para cumplir con el objetivo se utilizan los siguientes elementos:

- Device: Hace referencia al GPU (conjunto de multiprocesadores).
- Multiprocesador: Conjunto de procesadores y memoria compartida.
- Kernel: Función que tiene el código para ser ejecutada en la GPU.
- Grid: Conjunto de bloques.
- Bloques: Grupo de hilos SIMD.
- Hilos: Ejecutan el código de un kernel delimitando su dominio de datos según los valores de algunas variables (threadId, y blockId).
- Tamaño de warp: Es la resolución del planificador para emitir hilos por grupos a las unidades de ejecución.

La GPU ofrece a la CPU la visión de un coprocesador altamente ramificado en hilos, que tiene su propia memoria DRAM, donde los hilos se ejecutan en paralelo sobre los núcleos (*cores*) de un multiprocesador. Cada multiprocesador procesa lotes de bloques, uno detrás de otro; los registros y la memoria compartida de un multiprocesador se reparten entre sus hilos activos, los cuales se definen en una función llamada *kernel* la cuál será explicada más a detalle en las líneas que siguen.

Sintaxis. Elementos básicos.

CUDA extiende el lenguaje C con un nuevo tipo de función, *kernel*, que ejecuta en paralelo los hilos activos en GPU; el resto del código es C puro que se ejecuta sobre la CPU; así el típico **main()** de C combina la ejecución secuencial en CPU y paralela en GPU de kernels CUDA.
Un kernel se lanza de forma asíncrona, es decir el control regresa de forma inmediata a la CPU. Cada kernel GPU tiene una barrera implícita a su conclusión, lo que permite que no finalice hasta que no lo hagan todos sus hilos. Para distinguir las funciones que deben ser ejecutadas por la GPU se utilizan los siguiente modificadores:

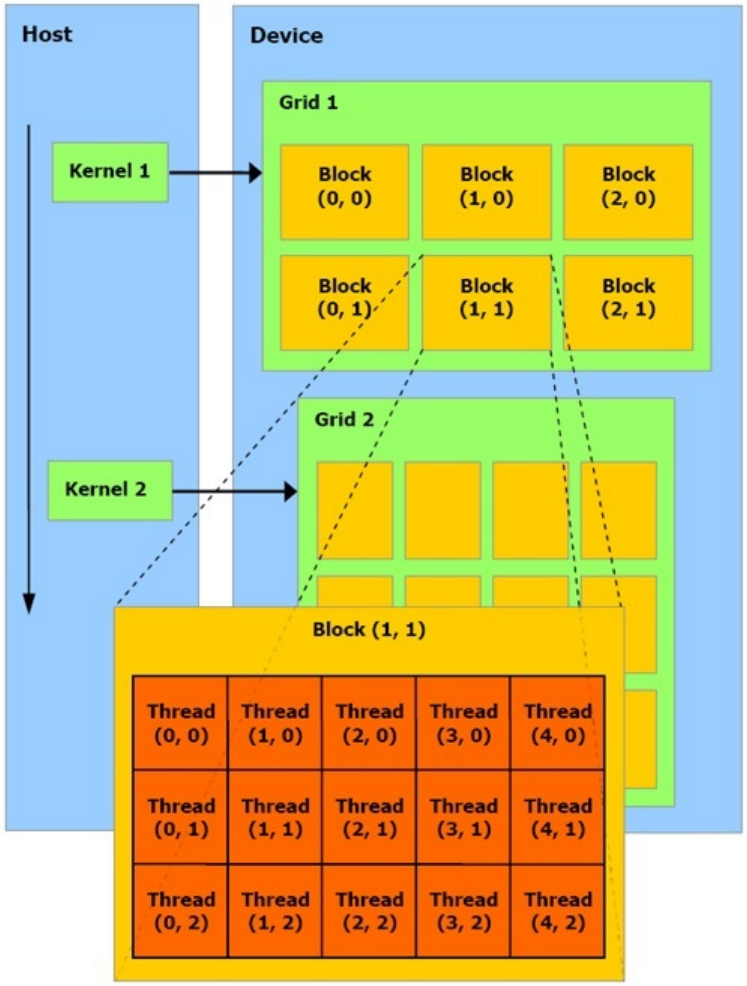


Figura 2.7: Representación de composición lógica.

- __**global**__ tipoRetorno nombreKernel() {}. Definen a las funciones kernel, que son llamadas desde la CPU.

- __**device**__ tipoRetorno nombreFuncion() { }. Definen funciones que solo pueden ser llamadas desde la GPU por las funciones kernel o desde alguna otra función de la GPU. Estas funciones no pueden tener llamadas recursivas ni llamadas indirectas a funciones mediante apuntadores.

- __**host**__ tipoRetorno nombreFuncion() { }. Definen a la funciones que solo pueden ser procesadas por la CPU, es decir una función simple de C.

Las palabras clave __host__ y __device__ pueden ser utilizadas simultáneamente en la declaración de una función, lo cual hace que el compilador genere dos versiones de la mis-

ma función, la versión para ser ejecutada por la CPU y la versión para la GPU.

La función kernel se define de la siguiente manera:

$$nombreKernel <<< gridDim, blockDim >>>$$

donde, gridDim es equivalente al número de bloques por Grid, y blockDim hace referencia al número de hilos por cada bloque, el cual debe estar en múltiplos de 32, por razones de eficiencia.

Como ya se mencionó, tanto la CPU y la GPU cuentan con sus espacios de memoria completamente separados, es por ello que para ejecutar una función kernel en un dispositivos GPU,es necesario realizar los siguiente:

1. Reservar memoria haciendo uso de las funciones, malloc(tamañoApuntador) para CPU y cudaMalloc (apuntador, tamaño) para GPU.

2. Transferir los datos necesarios desde el procesador principal hasta el espacio de memoria asignado al dispositivo, utilizando la función cudaMemcpy(*variableDestino*, *variableOrigen*, *tamañoVariable*, *sentenciaQueIndicaDirección*).

3. Invocar la ejecución de la función kernel deseada.

4. Transferir los datos resultado del procesamiento del kernel al CPU.

5. Al final del programa se debe liberar la memoria reservada, haciendo uso de las funciones free(puntero) para las variables de la CPU y cudaFree(apuntador) para las variables en la GPU.

A su vez existen cuatro distintas opciones para el parámetro *sentenciaQueIndicaDirección* las cuales son:

1. cudaMemcpyHostToHost: refiere que los datos se pasarán de la memoria de CPU hacia la memoria del mismo CPU.

2. cudaMemcpyHostToDevice: refiere que los datos se pasarán de la memoria de CPU hacia la memoria de la GPU.

3. cudaMemcpyDeviceToHost: refiere que los datos se pasarán de la memoria de la GPU hacia la memoria del CPU.

4. cudaMemcpyDeviceToDevice: refiere que los datos se pasarán de la memoria de la GPU hacia la memoria de la misma GPU.

Es importante tomar en cuenta que esta función permite realizar copias de datos de la memoria de un mismo dispositivo, pero no entre diferentes dispositivos.

2.7.2. Otros modelos paralelos: MapReduce

Sistema de Archivos Distribuidos (Distributed file system)

La mayoría de las equipos de cómputo están compuestos por un procesador, con memoria caché y un disco local; anteriormente las aplicaciones de procesamiento paralelo eran hechas en computadoras de propósitos específicos con muchos procesadores y hardware especializado. Sin embargo, debido a los requerimientos de las aplicaciones de hoy en día se ha vuelto necesario utilizar infraestructuras de cómputo conformadas por cientos de nodos (computadoras), lo cual reduce el costo comparado con máquinas de propósitos especiales.

Es importante explicar cuál es la organización física de los nodos de cómputo, misma que es llamada comúnmente como clúster de cómputo (cluster computing). Los nodos de cómputo se encuentran almacenados en *racks*, entre 8 a 64 en un solo rack. Los nodos en un rack están conectados por una red Ethernet de un gigabit. Hay muchos racks de nodos, los cuales están conectados en otro nivel de la red o por un *switch*. El ancho de banda entre racks es mayor que el de Ethernet pero dado el número de pares de nodos que pueden necesitar comunicarse entre racks, este ancho de banda puede jugar un papel esencial (ver figura 2.8).

Para trabajar con un clúster y poder aprovechar las ventajas de su organización los archivos deben comportarse de forma diferente a como lo hacen los sistemas de archivos convencionales, este nuevo sistema de archivos es llamado Sistema de Archivos Distribuido o DFS (Distributed File System) los cuales deben tener las siguientes características:

- Los archivos deben ser enormes, posiblemente de un terabyte de tamaño (DFS no fue creado para archivos pequeños).
- Los archivos raramente son actualizados. Los archivos deben ser de consulta o quizá se incremente la información que contenga pero con muy poca probabilidad.

Los archivos son divididos en *chunks*, los cuales son de 64 megabytes aproximadamente. Los *chunks* son replicados, x número de veces en x diferentes nodos; para que no se pierda

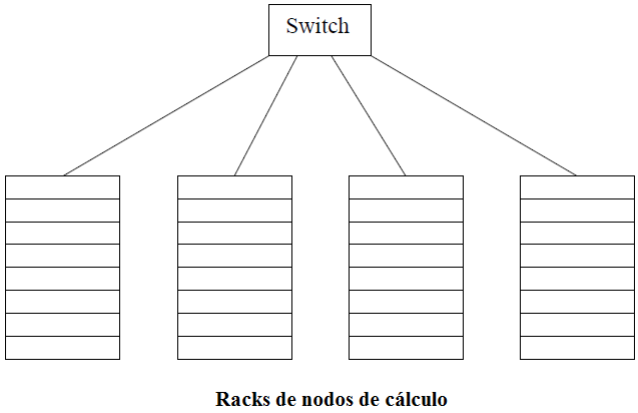


Figura 2.8: Organización física de nodos.

la información por completo si es que un rack llega a fallar. El tamaño de cada *chunk* y así como el grado de replicación es decidido por el usuario.

Para localizar los chunks que fueron creados a partir de un archivo, existe un archivo pequeño llamado *nodo maestro* (master node) o *nodo nombre* (name node). El nodo maestro se replica, y un directorio del sistema de archivos conoce donde se encuentran cada una de las copias. El directorio también es replicado y todos los integrantes del DFS conocen donde se encuentran dichas réplicas. [26]

Existen distintos sistemas de archivos distribuidos, los más usados son:

- The Google File System (GFS).
- Hadoop Distributed File System (HDFS), una versión open-source utilizado por Hadoop, una implementación de MapReduce (ver siguiente sección) y distribuido por Apache Software Foundation.
- CloudStore, un open-source desarrollado por Kosmix.

MapReduce

MapReduce es un modelo de programación y una implementación asociada para procesar y generar grandes conjuntos de datos, basado en el uso de dos funciones: *map* y *reduce*, tolerante a fallos por su sistema de replicación de la información.

Fue creado por Google donde es utilizado principalmente para el cálculo del PageRank (el cual es un conjunto de algoritmos utilizados para asignar de forma númerica la relevancia de las páginas web indexadas por un motor de búsqueda). Una de sus implementaciónes en código abierto más importantes es **Hadoop** la cual trabaja con el sistema de archivos distribuidos desarrollado por Apache Foundation (HDFS).

La función *Map*, escrita por el usuario, toma un par llave-valor del conjunto de entrada y produce un conjunto de pares de llaves/valores intermedios. La librería de MapReduce agrupa a todos los valores asociados con la misma llave y pasa los datos ya agrupados a la función *Reduce*. La función *Reduce* también debe ser escrita por el usuario; acepta como entrada una llave intermedia con el conjunto de valores relacionados a ella; incorpora esos valores para formar un conjunto de valores lo más pequeño posible. [26]

Es posible describir el funcionamiento de MapReduce de la siguiente forma:

1. Existen cierto número de tareas Map, a cada una se le da uno o más chunks de información del Sistema de archivos distribuidos. Dichas tareas Map transforman la entrada en una secuencia de pares llave-valor (*key-value*) intermedias.

2. Los pares llave-valor procedente de las tareas Map son recolectados por un *master controller* y ordenados de acuerdo a las llaves. Las llaves son divididas entre las tareas *Reduce*, de forma que todos los pares llave-valor que tengan en común la llave terminan en la misma tarea *Reduce*.

3. Las tareas Reduce en una llave a la vez, y combinan todos los valores asociados a dicha llave.

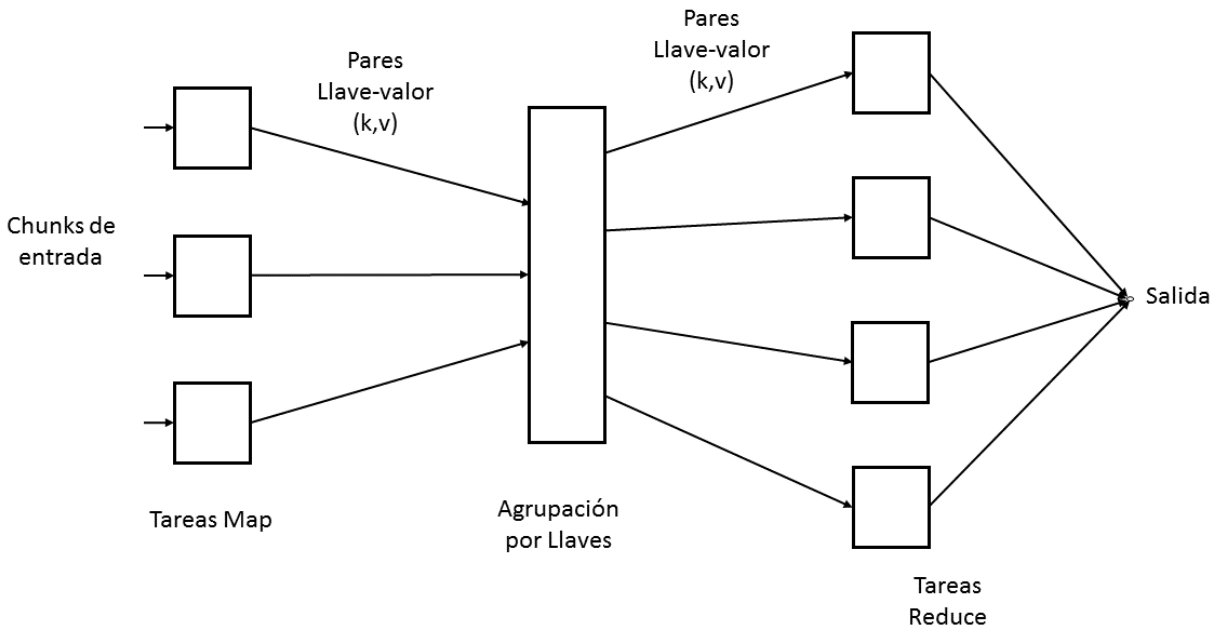


Figura 2.9: Esquema de cómputo de MapReduce.

Capítulo 3

Trabajos relacionados.

3.1. Importancia del paralelismo en aprendizaje de máquina

CUDA es una arquitectura y un modelo de programación que basa su funcionamiento en el modelo SIMD (Single Instruction, Multiple Data) es decir la misma instrucción es ejecutada en todos los procesadores con diferentes datos, lo cual es de gran utilidad cuando hablamos de algoritmos que manejan operaciones matriciales.

MapReduce es una arquitectura y modelo de programación que utiliza un poco del mismo concepto, un par de instrucciones ejecutadas en distintos procesadores con diferentes datos, sin embargo éste nos ofrece tolerancia a fallos, puesto que dichos procesadores se encuentran distribuidos en un clúster físico y hace uso de un sistema de archivos que permiten la replicación y distribución de la información.

Una de las ventajas que comparten ambos modelos es que se pueden implementar en distintos lenguajes de programación, por su parte CUDA se desarrolla sobre una extensión del lenguaje C, aunque también es posible trabajar sobre esta arquitectura desde otros lenguajes de programación como Python, Fortran, etc., a través de APIs que hacen más flexible la implementación. Como ejemplo está PyCuda que nos ofrece las ventajas de Python y sus bibliotecas que facilitan el manejo de matrices. El costo de esta ventaja se refleja en el tiempo de procesamiento, pues un kernel no puede ser escrito utilizando la sintaxis de python, forzosamente un kernel se debe programar utilizando la sintaxis C.

Aprendizaje máquina ofrece una amplia gama de algoritmos estadísticos para el análisis, la minería y la predicción de datos; incluye varias técnicas como árboles de decisión,

regresión, máquinas de vector de soporte y otras técnicas de minería de datos. Todos estos algoritmos son costosos desde el punto de vista computacional, lo que los convierte en casos ideales para implementarlos utilizando métodos de arquitectura y programación paralela.

Uno de los primeros esfuerzos en esta dirección se remonta a 1995, donde K. Thearling [42] discutió las posibilidades de mejorar el rendimiento de los enfoques populares de aprendizaje automático, como el razonamiento basado en la memoria, redes neuronales y algoritmos genéticos mediante la adopción de un procesamiento paralelo.

Existe una serie de configuraciones de los datos, sobre los que se decide escalar una tarea de aprendizaje máquina y utilizar alguna técnica de paralelismo.

1. **Tamaño de entrada.** En muchas áreas, el tamaño de la información generada es extremadamente grande, i.e. genética, biología, medicina, etc.

2. **Dimensiones de los datos.** La información suele caracterizarse por vectores que representan a cada uno de los datos. Dichos vectores son n-dimensionales, donde n es el número de rasgos que los conforman.

3. **Complejidad de modelos y algoritmos.** Los algoritmos de aprendizaje de máquina de alta precisión, tienen también, un alto grado de complejidad, modelos no lineales o que emplean funciones costosas desde el punto de vista computacional.

Las arquitecturas paralelas modernas se encuentran basadas en topologías híbridas dónde las unidades de procesamiento son organizadas en jerarquías con múltiples capas de memoria compartida. Por ejemplo los GPUs típicamente se conforman por docenas de multiprocesadores, cada una de ellas tiene múltiples hilos (procesos) organizados en bloques. Cada bloque tiene acceso a una memoria compartida local relativamente pequeña, y a una memoria global mucho más grande (con una latencia grande).

A diferencia de las arquitecturas paralelas, las plataformas de cómputo distribuido típicamente tienen largas distancias (físicas) entre los procesadores que las componen, lo cual implica latencias grandes y anchos de banda pequeños. Sin embargo los procesadores individuales pueden ser heterogéneos y la comunicación directa entre ellos puede ser limitada o inexistente. Ya sea vía envío de mensajes o por una memoria compartida, donde el caso extremo se puede presentar cuando todo el flujo de datos se limita a los alcances de la tarea, como en el caso de MapReduce.

La variedad de plataformas paralelas y distribuidas e infraestructuras disponibles para realizar tareas de aprendizaje de máquina puede parecer extensa. Sin embargo, las siguientes observaciones captan los aspectos clave de diferenciación entre las plataformas:

- **Granularidad de paralelismo.** Empleada en soluciones de hardware especifico (como los GPUs) que permiten el paralelismo de *grano fino* sobre datos y tareas, donde tareas elementales (operaciones en vectores, matrices y tensores) puede extenderse a través de múltiples procesadores con un rendimiento muy alto. Sin embargo utilizar esta habilidad requiere redefinir el algoritmo por completo, como un flujo de datos de tareas elementales y tratando de eliminar el cuello de botella. Con clústeres se pueden definir tareas de alta granularidad a costa de incrementar los costos de comunicación.

- **Grado de personalización de un algoritmo.** Dependiendo de la plataforma elegida, la complejidad del rediseño de algoritmos requerida para habilitar la simultaneidad puede variar desde el simple uso de soluciones de terceros para paralelizar automática una implementación imperativa o declarativa existente, hasta tener que volver a crear el algoritmo o incluso implementarlo directamente en hardware. Por lo general, la implementación de algoritmos de aprendizaje en plataformas de hardware específico (GPUs) requiere una experiencia significativa, una configuración de tareas segura y evitar ciertos patrones comunes de software como la ramificación. En contraste, los sistemas paralelos y distribuidos de nivel superior permiten el uso de múltiples lenguajes de programación comunes extendidos por APIs que permiten el paralelismo.

- **Escalar el conjunto de datos.** Las aplicaciones que procesan conjuntos de datos demasiado grandes para que se ajusten a la memoria normalmente se basan en sistemas de archivos distribuidos o en clústeres de memoria compartida. Los frameworks de programación paralela que están estrechamente acoplados con el almacenamiento de conjuntos de datos distribuidos permiten optimizar la asignación de tareas durante la programación para maximizar los flujos de datos locales. En contraste, la programación en el hardware de uso específico es independiente del almacenamiento de los datos, utilizados para conjuntos de datos grandes y requieren procesamientos manuales para maximizar el performance.

- **Ejecuciones fuera de línea y en línea.** Las plataformas distribuidas normalmente asumen que su usuario tiene una tolerancia más alta a fallos y la latencia en comparación con las soluciones de hardware específico. Por ejemplo, un algoritmo

implementado a través de MapReduce y enviado a un clúster virtual normalmente no tiene garantías sobre el tiempo de finalización. Por el contrario, los algoritmos basados en GPUs pueden asumir el uso dedicado de la plataforma, lo que lo hace preferible cuando de aplicaciones en tiempo real pensamos.

Las métricas utilizadas para el análisis del rendimiento de los algoritmos paralelos son las siguientes:

- **Speed-up** es la proporción del tiempo de ejecución entre la versión secuencial y la versión paralela de un algoritmo.
- **Eficiencia** mide la proporción del speed-up y el número de procesadores.
- **Escalabilidad** controla la eficiencia como una función de un número creciente de procesadores

Se utilizan a modo de comparación solamente, y proveen suficiente información en estudios empíricos como éste. Desde un punto de vista práctico, dadas las diferencias de hardware empleadas para implementaciones de las versiones paralelas y secuenciales, ver estas métricas como funciones de costos (hardware e implementación) es importante para comparaciones justas.

El siguiente paso natural es trabajar en entornos distribuidos como GPU o clústeres GPU / CPU y la nube utilizando un marco distribuido para MapReduce como Hadoop. En un sentido práctico, las GPU se han comportado mejor que las multinucleo y son mucho menos costosas. En el contexto de la nube también, donde el costo del hardware es una preocupación, tiene sentido adoptar GPUs para atender las preocupaciones de costo y velocidad.

El gran análisis de datos está ganando prominencia hoy en día. Aunque las técnicas para manejar el tamaño de los datos son maduras, las técnicas para derivar la semántica de una gran cantidad de datos están notablemente ausentes. Es muy importante invertir esfuerzos en sacar a la luz la semántica de los datos, donde la minería de datos y los enfoques paralelos tienen un gran papel que desempeñar. Los poderes de extracción del conocimiento del aprendizaje de máquina necesitan emplearse para aprender de los datos grandes y no estructurados de una manera distribuida. Esta es otra dirección todavía inexplorada. Este es un gran paso hacia la realización de la web semántica y la extracción de conocimiento mediante la minería de las fuentes no estructuradas.

A continuación se describen algunos de los trabajos que se han realizado con fin de paralelizar algoritmos de aprendizaje de máquina. Al inicio se encuentran enfoques que no tienen relación con GPUs ni con MapReduce

3.2. Minería de reglas de asociación y conjuntos de artículos frecuentes.

La minería de reglas de asociación (ARM) y los conjuntos de item frecuentes (FIM) por sus siglas en inglés son dos temas que están ampliamente relacionados. FIM se considera como un prerrequisito para ARM incluso se considera el paso más crítico dentro de la minería de reglas de asociación. Se define un ARM como el problema de llegar a la regla de la forma $A \Rightarrow B$ (A implica B), donde A y B son conjuntos de artículos frecuentes. El soporte de una regla se define como la probabilidad conjunta de transacciones que contienen tanto A como B y la confianza de la regla es la probabilidad condicional de que una transacción contenga B dado que contiene A. Zaki en [47] cubre todos los aspectos técnicos en cuanto a la paralelización de estas técnicas con un resumen de los esfuerzos similares entre 1996 y 1999.

Cubren todos los aspectos técnicos de la minería de reglas de asociación paralela y de la minería de artículos frecuentes. Se resumen los esfuerzos similares entre 1996 y 1999 y plantea cuestiones que varían de los tipos de algoritmo a características de los algoritmos. El presente informe examina casi todos los aspectos de la minería de datos, aunque no rastrea la contribución individual de los informes. Este es uno de los informes de referencia que resumen los esfuerzos hasta 1999. Mueller publicó un informe sobre la comparación de los algoritmos rápidos de minería de conjuntos de elementos secuenciales con enfoques paralelos[34] en 1995 que resume el estado del arte.

En este capítulo se realiza un resumen sobre los distintos trabajos relacionados con aprendizaje de máquina implementados tanto en GPUs como en clúster bajo el modelo de MapReduce.

3.3. Trabajos realizados en GPUs

A partir del año 2000 utilizar procesadores gráficos para tareas computacionales ha sido una de las principales tendencias dentro del cómputo paralelo. Sin embargo fue hasta principios de 2004 cuando el campo del aprendizaje de máquina comenzó a prestar atención a los GPUs. Aunque los años iniciales hasta 2004 no vieron la adaptación de la arquitectura de la GPU para el aprendizaje de máquina como una tendencia importante, los últimos años vieron algunos de los trabajos muy significativos en el aprendizaje de máquina y la minería de datos.

En esta sección se presenta un resumen del trabajo que contribuyó indirectamente a la investigación del aprendizaje de máquina. La asignación rápida de cadenas, operaciones con matrices, minería de flujos, búsqueda, clasificación y operaciones dentro de base de datos son problemas típicos abordados en este campo. La manipulación de matrices es una parte muy importante del procesamiento de imágenes, el cual a su vez es uno de los problemas habitualmente desarrollados en las GPUs. Mientras que algunos hicieron énfasis en la manipulación rápida de la matriz para el procesamiento de la visualización de imágenes, ciertos esfuerzos se enfocaron en la manipulación de la matriz para los métodos numéricos, que son útiles desde el punto de vista del aprendizaje de la máquina.

Entre las aplicaciones de computación de uso general de GPUs, el aprendizaje automático es de los más sobresalientes. Dado que las GPUs sólo entienden datos gráficos, los datos relacionados con problemas de computación de propósito general deben expresarse en términos de datos gráficos para ser resueltos. Se observa que en los años iniciales, se implementaron las funcionalidades básicas como la multiplicación matricial y otras operaciones de matrices, ordenación de cadenas, clasificación, procesamiento de consultas y otras operaciones de bases de datos, etc., necesarias para ejecutar los algoritmos complejos. Estos esfuerzos son un paso hacia la adopción del aprendizaje de máquina. Es interesante observar que una gran cantidad de trabajo que abarca una amplia gama de algoritmos se ha hecho en un lapso corto.

Uno de los primeros trabajos con procesadores gráficos estrechamente relacionados con el aprendizaje automático es el intento de ejecutar la multiplicación matricial, un elemento básico de cálculo de la mayoría de los algoritmos de aprendizaje de la máquina [43]. En el presente informe, los autores intentan adaptar la técnica del cálculo paralelo y realizar un cálculo de la porción en cada procesador. De hecho, este fue el primer intento de utilizar el

procesador gráfico para el cálculo general, cuando se creía que las GPUs no están pensadas para tales operaciones. Se demostró que las GPUs daban un rendimiento competitivo y desencadenó muchos intentos en esa dirección.

Un gradiente conjugado y resolver un multi-grid son problemas que se implementaron en una GPU en [4] y se demostró que las GPUs se pueden utilizar con éxito como un procesador de streaming con alto rendimiento en punto flotante. En el año 2009, un intento de acelerar la comparación de cadenas en GPU fue presentado por Schats [39]. Esto intenta marcar el comienzo del procesamiento de texto en GPUs.

El aprendizaje de máquina ofrece una amplia gama de algoritmos estadísticos para el análisis, la minería y la predicción de datos; incluye varias técnicas como árboles de decisión, regresión, máquinas de vector de soporte y otras técnicas de minería de datos. Todos estos algoritmos son costosos desde el punto de vista computacional, lo que los convierte en casos ideales para implementarlos utilizando métodos de arquitectura y programación paralela.

Entre los primeros algoritmos de ordenamiento a implementar en una GPU se encuentra el trabajo presentado por [18], donde se presentó un algoritmo de clasificación eficiente de caché que se puede utilizar en contextos de bases de datos y de minería de datos. Los autores usaron las capacidades de mapeo y mezcla de texturas de las GPUs para ordenamiento. El paralelismo y las capacidades de procesamiento de vectores de las GPU se explotaron para establecer esto. *Fat string matching* algoritmo dado por [39] es uno de los primeros intentos de utilizar GPUs para el procesamiento de cadenas.

Fang en [**govindaraju2005fast**] presentó un algoritmo rápido y aproximado de minería de flujo para la construcción de e-aproximación cuantil y resúmenes de frecuencia. Utilizaron ordenamiento como el elemento computacional para la construcción del histograma. El algoritmo de ordenamiento desarrollado en este contexto utilizó la red de clasificación periódica equilibrada y explotó la alta potencia computacional y el ancho de banda de memoria de las GPU. Este es uno de los primeros esfuerzos donde los datos masivos recolectados por registros, redes de sensores y seguimiento web se procesan como flujos generados por consultas periódicas y para calcular estadísticas numéricas.

Guha [20] presentó un seminario informativo sobre la utilización de la destreza de las GPU para la minería de datos y visualización para KDD audiencia. Este tutorial presentó

los métodos de explotación de la arquitectura paralela de las GPUs para algoritmos clásicos de agrupación y clasificación, regresión y otros análisis estadísticos típicos, operaciones de matriz / vector, flujo de datos, análisis y visualización. También familiarizó al público con las herramientas y aplicaciones para procesar datos como flujos con paralelismo de nivel de pixeles en GPUs.

" Agrupamiento escalable usando procesadores de gráficos por Cao [5] ". Es el primero en modificar un algoritmo de agrupación para GPU, haciéndolo escalable para procesar grandes datos. Los autores identifican los componentes de los algoritmos de agrupamiento que son computacionalmente caros y los modifican adecuadamente para ejecutarse en procesadores gráficos. El trabajo presentado se basa en un algoritmo de K-medias, donde la computación a distancia y la comparación son operaciones costosas. Los autores proponen un nuevo esquema de medición de distancia basado en el procesamiento de vectores de fragmentos y el procesamiento de múltiples pasadas. El algoritmo minimiza la transmisión de datos entre la CPU y la GPU teniendo en cuenta el bajo ancho de banda.

Un nuevo algoritmo de agrupamiento de documentos inspirado en la naturaleza llamado "el algoritmo basado en flocking" se presenta en el informe de Charles [10] en el Journal of Undergraduate Research. En este enfoque basado en agentes, la organización del documento surge a través de una interacción entre un grupo de agentes. Los documentos similares se reúnen y se organizan libremente según el tema. Los autores implementaron y probaron el algoritmo en plataformas secuenciales y paralelas y compararon los resultados. Aunque la GPU superó el rendimiento de procesador único, el método planteaba ciertas restricciones sobre el número de documentos que se podían procesar.

GPUTeraSort, fue una aplicación que realizó la clasificación de miles de millones de registros de bases de datos fue presentada por Govindaraju [19]. Este es uno de los informes que impulsaron los esfuerzos computacionales a gran escala en las GPU. El algoritmo de ordenamiento utilizó tanto los datos como el paralelismo de tareas y logró un alto rendimiento en la GPU y mostró que terabytes de datos pueden procesarse a costa de unos cuantos centavos. En el año 2007 también, un esfuerzo de procesamiento de consulta de base de datos en GPU fue presentado en [**fang2007gpuqp**]. El mismo grupo de investigación presentó otro informe en el próximo ACM SIGMOD en 2008 sobre operaciones relacionales en GPU. Estos tres informes están marcados como hitos en el procesamiento de bases de datos en GPUs. Este grupo comenzó su trabajo en las GPU ya en 2005, donde presentaron algoritmo de ordenamiento eficiente para las operaciones de base de datos.

Otro informe presentado en el Journal of Parallel and Distributed Computing escrito por Che [11]. Discutió un salto de rendimiento logrado mediante la implementación del algoritmo de agrupación de K-medias en GPUs entre las otras aplicaciones de propósito general de GPU.

GPUminer presentado por Fang [15]. Discutieron un sistema paralelo de minería de datos que implementaron K-medias y algoritmos de minería de patrones frecuentes apriori. Aunque los autores no compararon el rendimiento de su algoritmo de K-medias con las implementaciones anteriores, presentaron los estudios de rendimiento en una CPU *quad-core* y GPU. Este intento se centró en la aplicación donde el almacenamiento de datos y la gestión de búfer se basa en la CPU, la minería paralela se basa en co-procesamiento y la visualización se basa en GPU.

Un método de entrenamiento y predicción de SVM fue implementado por A. Carpenter y fue presentado en [6]. La heurística de segunda orden empleada reduce significativamente las iteraciones requeridas para ejecutar el módulo para descomponer y resolver el programa cuadrático. El método resulta ser eficiente a medida que aumenta el número de puntos de datos. Sorprendentemente, este intento precedido por Catanzaro [8], que utilizó el paradigma MapReduce para desarrollar la solución para SVM. Esto usó CUDA, el lenguaje especializado para programar en GPU.

El Análisis Semántico Latente, una técnica usualmente utilizada para reducir la dimensión de los conjuntos de datos de términos de documentos usando la composición de valores singulares (SVD) fue diseñada para funcionar en GPU por Cavanagh [9]. Ellos eligieron el algoritmo Lancozs que tridiagonaliza una matriz para computar SVD más rápido, ya que implica un vector matricial. Este intento fue también una de las primeras implementaciones de CUDA. Las limitaciones del sistema y el lenguaje utilizado fueron las limitaciones para aprovechar al máximo los beneficios de tal implementación.

Un esfuerzo para optimizar el algoritmo de agrupamiento de K-medias se presentó en Zechner [48]; Esta vez aprovechando los beneficios de la codificación en CUDA. Ese informe aprovechó los aspectos arquitectónicos de la plataforma para superar las deficiencias de las implementaciones anteriores. Los cálculos de distancia fueron paralelizados en GPU y la actualización secuencial de centroides se realizó en una CPU.

Un esfuerzo para implementar el algoritmo Apriori para la minería de articulos frecuentes (FIM) en hardware GPU se mostró en Fang [16]. Los autores demostraron dos formas de implementar la FIM; uno de los cuales usó la GPU sola mientras que el otro utiliza tanto la GPU como la memoria de la CPU. Utilizaron un enfoque basado en mapa de bits que utilizó un trie para demostrar ambos enfoques mencionados anteriormente.

Construir la representación de TF / IDF a partir de datos de texto sin procesar es una tarea importante involucrada en la minería de texto. El problema de clasificar documentos basados en la búsqueda de TFIDF fue acelerado usando GPU por Zhang y Potok [43]. El informe fue uno de los primeros para trabajar en la minería de textos. Se trató principalmente de introducir el paralelismo en el proceso de construcción de una tabla hash de frecuencia simbólica para cada documento. Esto requiere que las tareas de preprocesamiento como tokenizing, stemming, etc., sean ejecutadas de manera paralela. Entonces se construye una tabla de hash de frecuencia de token global y se calculan los valores de TF / IDF para encontrar la relevancia de un documento con respecto a un tema.

De acuerdo con [45], uno puede lograr una buena cantidad de éxito en el procesamiento de los conjuntos de datos muy grandes que no encajan en la memoria GPU. Aquí, los autores investigaron principalmente la viabilidad de procesar conjuntos de datos muy grandes en GPU para implementaciones prácticas. Ellos demostraron el popular algoritmo de agrupamiento de K-medias y midieron el desempeño contra el benchmark 'Minebench' para probar la eficiencia de su enfoque.

En 2010, el problema de la agrupación de datos se abordó en tres informes [13], [50] y [49]. Es interesante observar que estas contribuciones que aparecieron en revistas / conferencias conocidas provienen del mismo grupo. La solución de agrupación de datos a gran escala presentado en [13] se basó en un algoritmo de flocado mencionado en uno de sus trabajos anteriores [10]. En el cuál se describen métodos para superar los déficit experimentados con el trabajo anterior, donde el gran tamaño de la información del documento causaba frecuentes lecturas de la memoria global del dispositivo GPU. El dispositivo de memoria global no se almacena en caché y tiene un retardo de cientos de ciclos por lectura. Para encontrar este problema, algunos términos de documento se almacenaron en caché en memoria compartida para un acceso más rápido. La aproximación inicial también fue reemplazada por un nuevo método que utilizó una matriz de similitud de documentos que podría utilizarse para leer directamente para la comparación de documentos.

El algoritmo de floculación utilizado en [10] no se considera como una de las técnicas de aprendizaje de máquina; sin embargo; el propósito del agrupamiento de documentos se suele abordar mediante la minería de texto, un enfoque de aprendizaje de máquina. El mismo grupo de investigación demostró un enfoque altamente paralelizado para calcular los valores de TF / IDF para reunir la similitud de documentos como en la minería de textos [43]. Emplearon el algoritmo de simulación basado en flocking para el agrupamiento de documentos cantar los valores TF / IDF para la comparación de similitud en [50] y mostró un aumento de rendimiento. Otra medida de similitud TF / ICF propuesto en otro artículo fue empleado también aquí como el cálculo de TF / IDF en la arquitectura paralela no mejoró el rendimiento en gran medida. En resumen, archivó varios enfoques y combinó aspectos de aprendizaje de máquina con algoritmos de simulación y sugirió alternativas para el agrupamiento de documentos.

La mejor combinación de resultados en el esfuerzo mencionado anteriormente; el agrupamiento basado en la agrupación similaridad medición calculada utilizando el TF / ICF se implementó en un nodo cuatro GPU grupo y se presentó en [49]. Los algoritmos ML basados en métodos de núcleo varían poco a medida que aumenta el tamaño de los datos. Este problema fue tratado por Srinivasan [41] empleando GPU parcialmente. Las GPU se utilizan típicamente para acelerar las operaciones de la matriz tales como la descomposición, el producto y sus formulaciones iterativas, que se utilizan ampliamente en los métodos de núcleo.

Tres intentos por implementar de árboles de decisión (de diferente naturaleza) en las GPU se presentan en [40], [37], [17]. El informe presentado en [40] demostró un motivo de reconocimiento de objetos. Desarrollaron una estrategia para transformar la estructura de datos forestales de una lista de árboles binarios a una textura 2D. La implementación fue cien veces más rápida en comparación con la contraparte de la CPU. Una agrupación de contexto basado en el árbol se demuestra en [37]. La división de nodos en el árbol de decisión se lleva a cabo independientemente de todas las demás divisiones, ganando una velocidad considerable. Un árbol de paralelismo en el nivel se adoptó en [17] para construir muy aleatorizado árboles.

Un intento de implementar una red neuronal sobre GPUs se presenta con Paprotski en [29].A diferencia de los enfoques de paralelización normal, este enfoque se centró en obtener las operaciones de matriz de elemento optimizado para que la ganancia global en el rendimiento se cumple. El informe mostró que la ganancia de implementación fue

alrededor de 66 veces, aunque las operaciones de elemento aceleraron más de 1000 veces, debido a la sobrecarga de comunicación.

El trabajo relatado por Gneuron [38] es otro intento de construir redes neuronales en las GPU, donde los autores hicieron un intento de identificar los hotspots que son críticos y trató de paralelizar el mismo. Se centró en los elementos de las operaciones dentro del ciclo de formación, pero el ciclo de formación no fue paralelo.

3.4. Trabajos realizados utilizando MapReduce

La técnica MapReduce popularizada por Google ha traído un cambio de paradigma a la forma en que los datos se manejan en entornos distribuidos / paralelos. Vemos la adopción de la técnica de reducción de mapa desde la era multicore. Claramente, con el advenimiento de las GPU, la técnica MapReduce resultó ser una herramienta eficaz para elevar el rendimiento paralelo.

La primer implementación de MapReduce el marco en un entorno multicore y la posterior demostración de un algoritmo de aprendizaje automático fue realizada por [12]. Los autores presentaron un marco de aplicación de MapReduce ampliamente aplicable para ejecutar la mayoría de los algoritmos de aprendizaje de la máquina. El marco se basa en la premisa de que todos los algoritmos pueden expresarse en forma de suma. Los cálculos son realizados por diferentes correlacionadores y la suma final es llevada a cabo por el reductor. Los autores demostraron el funcionamiento de LWLR, LR, Naïve Bayes, PCA, Máquina de Vector de Soporte, ICA, GDA, ICA y red Neural, etc.

Tanto en [21] como en [7] se desarrollaron trabajos de MapReduce en el marco de las GPUs utilizando CUDA. Mientras Catanzaro, demostró el uso de este marco implementando SVM en él. Funciones demostradas tales como relación de cadenas, multiplicación matricial, construcción de índices invertidos, etc. desarrollado un conjunto de APIs similares a los de un mapa basado en la CPU reducir la implementación y ocultó las complejidades de programación en la GPU considerablemente. El marco desarrollado por Catanzaro requiere que el usuario defina una función de map, un conjunto de operadores de reducción y una función de limpieza funcione sobre los resultados de la reducción. DisMarc es uno de los esfuerzos recientes que explora la destreza informática de las GPU para procesar enormes datos en un mapa de reducir el marco [33]. Con éxito oculta la complejidad de la

programación de GPU detrás de un simple MapReduce.

Un algoritmo paralelo FP Growth fue presentado por Li [27]. Un intento de paralelizar un FP Growth, un popular algoritmo FIM se hizo aquí. El algoritmo divide los cálculos de tal manera que cada máquina lleva a cabo un conjunto de tareas de minería independientes, eliminando las dependencias computacionales entre las máquinas. Se mostró una aceleración lineal. [28] publicó un libro sobre el procesamiento de texto intensivo de datos con mapa reducir en 2010. En este libro el autor proporciona una explicación detallada del sistema de MapReduce y el método para implementar varios enfoques de procesamiento de texto en ellos. Ellos discuten todo el enfoque sobre las arquitecturas distribuidas de Hadoop para enfatizar la ventaja paralela producida por MapReduce.

Un intento de implementar un conjunto masivamente paralelo de árboles de decisión con MapReduce se ha presentado en [36]. Un trabajo de reducción de mapa para encontrar la mejor división cuando hay demasiados datos para encajar en la memoria se definió. MapReduce también se empleó para cultivar el árbol entero, siempre y cuando se ajusta en la memoria.

Una observación importante es que la amplitud y profundidad de la investigación en el espacio ARM es muy alta y sigue un registro continuo. Esto parecía ser natural en cierto modo; como ARM se requiere para procesar enormes bases de datos e indicó la necesidad de adoptar enfoques paralelos para reducir el tiempo de procesamiento. Sin embargo, otros enfoques de aprendizaje de la máquina se han estudiado de forma intermitente, aunque los árboles de decisión y las redes neuronales parecen tener una parte justa también. Más que la innovación académica, los escenarios de aplicación parecen haber impulsado la investigación en este espacio. Es importante notar que hay un amplio margen para trabajar en estas áreas.

Se han realizado muchos trabajos en el frente teórico y la mayoría de los problemas de rendimiento están bien abordados. Sin embargo, la brecha más importante es en el frente de la aplicación, donde estos algoritmos se despliegan en contextos comerciales reales. El mundo de los negocios es testigo de muy pocos escenarios en los que los beneficios del aprendizaje paralelo de máquinas sean realidad. Con el advenimiento del entorno de la nube, existe la necesidad de que las aplicaciones de la vida real se modifiquen para el contexto de la nube. La necesidad de trabajar con una gran cantidad de datos debería abrir muchas oportunidades para implementar algoritmos paralelos.

MapReduce aún no ha sido completamente explotado en el contexto de aprendizaje de la máquina. Dado que MapReduce puede mejorar en gran medida el rendimiento de las aplicaciones en GPUs, tal combinación sería una bendición para el aprendizaje de la máquina. Esto haría que el análisis, la visualización de datos y la predicción se ejecuten en los escritorios sin latencia, independientemente de la cantidad de datos que deben tratarse. La mayor ventaja es la reducción del coste provocada por las GPU.

Capítulo 4

Metodología.

Dada la arquitectura de CUDA el trabajar con matrices se da de forma hasta cierto grado sencilla. Se eligió para este trabajo de tesis un algoritmo basado en una distribución de probabilidad: la versión EM de Mezcla de Gausianas, el cual representa un tiempo de ejecución alto debido a las operaciones matriciales que requiere. Para minimizar este tiempo se propone la combinación de dicho algoritmo con el algoritmo K-medias.

Este capítulo se divide de la siguiente manera:

- **Algoritmo implementado *Mezcla de Gaussianas*.** Se describe los detalles de la implementación del algoritmo Mezcla de Gaussianas.
- **Opciones de paralelismo.** Se describen las opciones de paralelismo que pueden ser aplicadas al algoritmo propuesto.
- **Justificación del modelo de paralelismo implementado.**

4.1. Algoritmo implementado, *Mezcla de Gaussianas*

Los algoritmos basados en distribuciones de probabilidad gaussianas son métodos comúnmente utilizados para la clasificación supervisada, sin embargo tienen limitaciones cuando se trata de problemas donde las clases no pueden ser linealmente separables. Una solución es combinar funciones de probabilidad. A esta técnica se le conoce como Modelo de Mezcla Finita (Finite Mixture Model) y es un enfoque ampliamente utilizado

porque es un método paramétrico que se puede aplicar a problemas de clasificación no lineal. Se puede definir a este modelo como:

$$p(x) = \sum_{j=i}^{g} \pi_j p(x, \Theta_j) \tag{4.1}$$

Donde, g es el número de componentes de la combinación de funciones de probabilidad; π_j es la probabilidad de los componentes, el cual también es conocido como el peso de cada componente, tal que

$$\sum_{j=1}^{g} \pi_j = 1 \tag{4.2}$$

y $p(x, \Theta_j)$ es la función de densidad de probabilidad con respecto a los parámetros Θ_j.

Por otro lado si la definición inicial de parámetros μ, σ^2 y π deja de ser aleatoria y se definen con base en el resultado de un proceso de agrupamiento previo, es posible acercarse a una solución óptima en menos tiempo.

El algoritmo que se propone es una combinación del algoritmo K-medias y la versión EM del algoritmo Mezcla de Gaussianas. En donde se aplica al conjunto de datos de entrada el algoritmo K-medias definiendo en un principio k como el número de clústeres en los que se desea dividir el conjunto de datos de entrada.

El primer paso del algoritmo K-medias es elegir aleatoriamente valores que representen a los K centroides. Es entonces cuando comienza el proceso iterativo; con los valores fijos de los centroides para cada uno de los datos se evalúa una función de similitud o distancia entre el dato, y cada uno de los centroides, para este trabajo se utilizó la distancia euclidiana:

$$dis(x_i, C_k) = \sqrt{\sum_{l=1}^{d} |x_{il} - C_{kl}|^2} \tag{4.3}$$

donde x_i representa al i-ésimo dato dentro del conjunto de datos, C_k representa al centroides respecto al que se está tomando la distancia para $k = 1, 2, ..., K$, d es la dimensión del vector que representa a un dato. Se asigna al dato al clúster representado por el

centroide con el cuál el valor de la función distancia sea el mínimo.

A continuación se fija la asignación de los datos dentro de los K clústeres y con base en ellos se recalculan los centroides, obteniendo el vector medio de cada clúster:

$$c_k^{nuevo} = \frac{1}{n_k} \sum_{j=1}^{n_k} x_{kj} \tag{4.4}$$

donde $k = 1, 2, ..., K$, n_k es el número de elementos que contiene el clúster k y x_{kj} es el j-ésimo elemento en el clúster k.

Para terminar la etapa iterativa de este algoritmo se proponen dos condiciones de paro:

1. Evaluar la función distancia entre los centroides actuales y los centroides nuevos, calculados durante la iteración. Si el valor que se obtenga de $dis(C_{actual}, C_{nuevo})$ es menor o igual que un valor definido desde el principio entonces el algoritmo habrá terminado.

2. Evaluar el número de iteraciones que ha realizado el algoritmo. Se indica un número de iteraciones límite, y se lleva un contador que indica cuando ese límite se alcance.

Para dar por terminada la parte iterativa, es necesario que al menos una de las condiciones de paro se cumplan, es importante mencionar que la evaluación se realiza en el orden que aquí se describe.

El resultado final se conforma por la división de los datos en K clústeres, cada uno con su respectivo centroide. Dichos resultados se utilizan para crear la entrada al algoritmo Mezcla de Gaussianas de la siguiente manera:

1. La asignación de elementos a los K clústeres se mantiene igual para el nuevo algoritmo. Es decir comenzamos la ejecución con los elementos ya agrupados.

2. Los parámetros μ, Σ y π:

   - **Media μ:** son representados por los centroides obtenidos.
   - **Matriz de covarianza Σ:** Se calculan para cada uno de los K clústeres, haciendo uso de la siguiente expresión:

$$\Sigma = \frac{1}{n_k} \sum_{i=1}^{N_k} (x_i - \mu)(x_i - \mu)' \tag{4.5}$$

donde n_k es el número de elementos en el k-clúster, x_i representa al i-ésimo elemento del clúster y μ es la media.

- **Coeficientes π:** es la probabilidad que tiene cada uno de los K clústeres de que un punto se encuentre dentro de ellos; y se calcula:

$$\pi_k = \frac{n_k}{N} \tag{4.6}$$

donde n_k es el número de elementos en el k-clúster y N es el número de datos de nuestro conjunto.

Una vez terminada la inicialización de los parámetros se calcula por cada uno de los elementos del conjunto de datos la probabilidad con la cual ese dato se encuentra asignado al clúster correspondiente, a través de la siguiente expresión:

$$\gamma(z_{nk}) = \frac{\pi_k \mathcal{N}(x_i|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(x_i|\mu_j, \Sigma_j)} \tag{4.7}$$

donde x_i es el i-ésimo elemento del clúster, π_k, μ_k y Σ_k son los parámetros correspondientes al mismo clúster. Es entonces cuando se procede la parte iterativa en donde se realizan los pasos E y M.

El **paso E** consiste en fijar los valores de los parámetros que definen la distribución de probabilidad y calcular las probabilidades posteori de cada uno de los datos, es decir la probabilidad con la que un dato se encuentra en cada uno de los K clústeres. Durante este paso se realiza la asignación de los datos a cada clúster, para cada uno de los datos:

- Se calcula la probabilidad con la que se asigna ese dato a cada clúster:

$$\gamma(z_{nk}) = \frac{\pi_k \mathcal{N}(x_i|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(x_i|\mu_j, \Sigma_j)} \tag{4.8}$$

  para $k = 1, 2, ..., K$

- Se realiza una comparación entre las K probabilidades calculadas y se elige la mayor, esta será la probabilidad posteori asignada al dato.

- Y el dato es asignado al clúster de los parámetros con los cuales se obtuvo la probabilidad anterior.

El **paso M** consiste en recalcular el valor para los parámetros π_k, μ_k y Σ_k representantes de cada uno de los K clústeres, a partir de mantener fijos los valores de las probabilidades posteori que se obtuvieron durante el paso E. Para ello se hace uso de las siguientes expresiones (descritas durante el capítulo 2):

- **Media μ_k^{nuevo}:**

$$\mu_k^{nuevo} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) x_n \tag{4.9}$$

- **Matriz de covarianza Σ_k^{nuevo}:**

$$\Sigma_k^{nuevo} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})(x_n - \mu_k^{nuevo})(x_n - \mu_k^{nuevo})^T \tag{4.10}$$

- **Coeficientes π_k^{nuevo}:**

$$\pi_k^{nuevo} = \frac{N_k}{N} \tag{4.11}$$

donde:

$$N_k = \sum_{n=1}^{N} \gamma(z_{nk}) \tag{4.12}$$

Para salir de la etapa iterativa es necesario definir la condición de paro, la cual quedó determinada por dos valores:

1. El cambio entre el logaritmo de la función de probabilidad de la iteración anterior y la iteración actual.
2. Número de iteraciones.

El algoritmo realiza la verificación en el orden dado, es decir, primero calcula (con los valores obtenidos durante la iteración actual) el valor de la siguiente expresión:

$$lnp(X|\mu, \Sigma, \pi) = \sum_{n=1}^{N} ln \sum_{k=1}^{K} \pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k) \tag{4.13}$$

y luego lo compara contra el valor calculado en la iteración anterior a través de:

$$diferencia = |lnp(X|\mu, \Sigma, \pi) - lnp(X|\mu^{anterior}, \Sigma^{anterior}, \pi^{anterior})| \tag{4.14}$$

Si la diferencia es menor o igual a cierto límite definido desde el inicio el algoritmo habrá terminado. Por el contrario, si la condición anterior no se cumple en la actual iteración se compara el valor de un contador que lleva el registro del número de iteraciones hechas, contra otro parámetro definido como límite de iteraciones; si el contador es mayor que ese límite entonces el algoritmo habrá terminado; asegurando con esto que el algoritmo sea finito.

En caso de que ninguna de las dos condiciones se cumplan se dará inicio a una iteración nueva, hasta que alguna de las dos condiciones sea verdadera. Para el caso de este trabajo, se definieron los limites de la siguiente manera

- Valor para diferencia entre parámetros actuales y nuevos en 0.00001.
- Número de iteraciones límite en 20,000.

4.2. Opciones de paralelismo

CUDA nos permite combinar el cómputo paralelo con el cómputo serial, es por ello que es importante definir cuáles son las etapas del algoritmo en las que es más conveniente aplicar paralelismo y en qué forma o con base en qué operaciones es más factible aplicarlo. A continuación se presenta el análisis sobre este tema que se realizó para el desarrollo de la programación del algoritmo propuesto.

Existen varios tipos de computación paralela:

- Paralelismo a nivel bit.
- Paralelismo a nivel de instrucción.
- Paralelismo de datos.
- Paralelismo de tareas.

En el caso de CUDA el paralelismo a explotar es a nivel datos, el cual se enfoca en la distribución de datos entre varios procesadores, las tareas que realizan son comunes y el resultado es uno solo. Como se comentó en capítulos anteriores las arquitecturas GPU siguen el modelo SIMD, una instrucción, múltiples datos.

El algoritmo cuenta con dos grandes etapas iterativas, podría en un principio pensarse que cada una de esas etapas iterativas se realizarán en paralelo, sin embargo debido a la naturaleza de los propios algoritmos esto resulta imposible pues para obtener los valores de cada iteración es necesario tener conocimiento de los valores calculados en iteraciones anteriores.

Otro enfoque de paralelismo se presenta con base en los parámetros que el algoritmo calcula a lo largo de su ciclo de vida, es decir podemos:

1. Paralelizar con base en K. Es decir en función del número de clústeres requeridos al inicio del algoritmo.

2. Paralelizar con base en N. Es decir en función del número de elementos que contiene el conjunto de datos de entrada.

Sin embargo el paralelizar el algoritmo completo sólo con base en K, resulta muy poco conveniente ya que a pesar de que el poder de procesamiento se incrementa de 1 a K, es posible incrementar ese número mucho más además de estar aprovechando muy poco las opciones que presenta CUDA; es necesario entonces identificar las operaciones que pueden paralelizarse a un grado K y las que pueden en grado N.

Es importante tener en consideración las restricciones del hardware que puedan presentarse, por ejemplo el uso de la memoria compartida de bloques puede resultar útil lógicamente, pero ser muy pequeña y por ende insuficiente para el propósito final.

4.3. Modelo de paralelismo implementado

Para la versión final del algoritmo propuesto se utiliza el modelo SIMD con base en los parámetros K y N dependiendo de las distintas etapas del algoritmo. A continuación se trata de explicar etapa por etapa la forma en la que se aplicó el paralelismo.

El primer paso fue paralelizar el algoritmo K-medias, para pre-procesar la entrada que tendría en algoritmo de mezclas gaussianas, dando por hecho la carga del conjunto de datos de entrada en memoria, tanto de la CPU como de la GPU, representado como una matriz de N elementos por M número de características que representan los datos; se prosigue con el siguiente procedimiento:

Algorithm 2 K-medias

Require: Conjunto de **N** datos, **K**, condición de paro 1 **STOP1**, y condición de paro 2 **STOP2**.

1: **GPU** Obtener los **K** centroides aleatorios; $z_1, z_2, ..., z_K$
2: **while** $dist(Z^{anterior}, Z^{nuevo}) > STOP1$ Y $contador < STOP2$ **do**
3: **GPU** Asignar el dato x_i a un clúster C_j; $i = 1, 2, ..., N$ y $j = 1, 2, ..., K$ para el cuál $dist(x_i, C_j)$ sea mínima.
4: **GPU**Calcular los nuevos centroides: $z_k^{nuevo} = \frac{1}{n_k} \sum_{j=1}^{n_k} x_{kj}$; $k = 1, 2, ..., K$
5: **end while**

Nota: Para saber hacer el cálculo de las distancias entre los centroides nuevos y los anteriores se realiza en paralelo la diferencia de los K centroides y en una bandera que nos indica si la diferencia en igual o mínima a la impuesta anteriormente.

Una vez que se obtienen el resultado de la ejecución de K-medias, esos resultado el procesamiento que se sigue es el siguiente:

Algorithm 3 K-Mezcla de Gaussianas. (lh = likelihood)

Require: **K**, conjunto de K clústeres y un vector medio por cada clúster, condición de paro 1 **STOP1**, y condición de paro 2 **STOP2**.

Ensure: K conjuntos de parámetros (μ, Σ) tal que $p(X|\mu, \Sigma)$ sea máxima.

1: **GPU** Obtener los **K** centroides aleatorios; $z_1, z_2, ..., z_K$
2: **while** $diferencia(log(lh)^{anterior}, log(lh)^{nuevo}) > STOP1$ Y $contador < STOP2$ **do**
3: **GPU** Paso E. Calcular la probabilidad para cada uno de los elementos del conjunto de datos.
4: **GPU** Paso M Re calcular los parámetros$\mu^{nuevo}, \Sigma^{nuevo}, \pi^{nuevo}$
5: **end while**

Se trató de paralelizar con base en las operaciones comunes para los elementos del conjunto de datos a tratar, utilizando la memoria global. En un principio se trató de utilizar la memoria compartida sin embargo el primer las primeras implementaciones mostraron problemas en cuanto al tiempo de ejecución aumentando el tiempo 20 veces más solo para un corrimiento de K-medias con una sola condición de paro, la cual fue de 100 iteraciones.

Capítulo 5

Experimentos y resultados

Se implementó el algoritmo de la forma descrita en el capítulo anterior de forma serial y en paralelo. En este capítulo se describen los datos con los que se realizaron los experimentos y se presenta de forma gráfica los resultados obtenidos.

5.1. Conjunto de datos

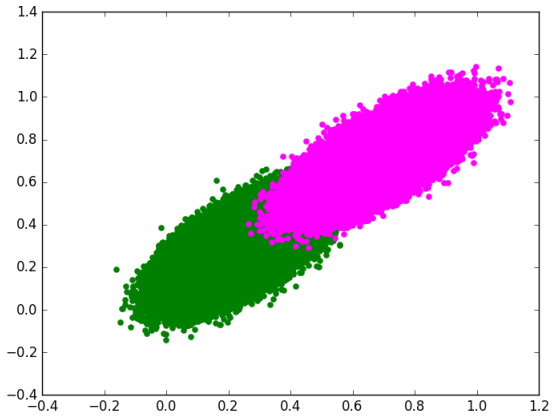
Con el fin de poder medir el rendimiento de nuestra implementación paralela y poder constatar que se está mejorando el tiempo de ejecución se realizó una pequeña aplicación en Python 3.5 para obtener conjuntos de datos de n elementos, que son producidos por un K número de distribuciones gaussianas que depende de los parámetros (μ, Σ) los cuáles también son asignados aleatoriamente. El flujo de la aplicación es el siguiente.

1. Se genera un número K aleatorio entre el número 2 y 10, esta variable representa el número de distribuciones gaussianas que vamos a simular.
2. Se generan los K vectores μ y las K matrices de covarianzas Σ teniendo en cuenta que dichas matrices deben ser simétricas.
3. Se definen los coeficientes pi, números decimales que representan la probabilidad de que cada punto sea generado por una de las K distribuciones gaussianas definidas por los parámetros del paso 2.
4. Con los parámetros que representan las K distribuciones gaussianas, se generan n cantidad de elementos en función de los coeficientes de probabilidad. Es decir, considerando a los coeficientes de probabilidad como el vector $\pi = (\pi_1, ..., \pi_l, ..., \pi_k)$, el

conjunto de datos resultado se forma por $n_1 = \pi_1 n$, ..., $n_l = \pi_l n$, ..., $n_k = \pi_k n$ donde $n = \sum_{l=1}^{k} n_l$.

Se obtuvieron conjuntos de datos de 100, 1000, 10000 y 100000 de elementos, cada uno con vectores de dimensiones 2, 3, 4, y 5; dichos datos fueron arrojado por las distribuciones gaussianas definidas por el software descrito anteriormente.

Para ejemplificar los conjuntos de datos obtenidos, se obtuvieron las representaciones gráficas de 3 de los conjuntos de datos. Dichas gráficas se muestran en un plano bidimensional; en la descripción se presenta el número de elementos y el valor de K, el cual hace referencia al número de distribuciones gaussianas que se utilizaron.



Parámetros 1

$\pi = 0.52$

$\mu = [\,0.25645, 0.2447]$

$\Sigma = \begin{bmatrix} 0.008233 & 0.005839 \\ 0.005839 & 0.00506 \end{bmatrix}$

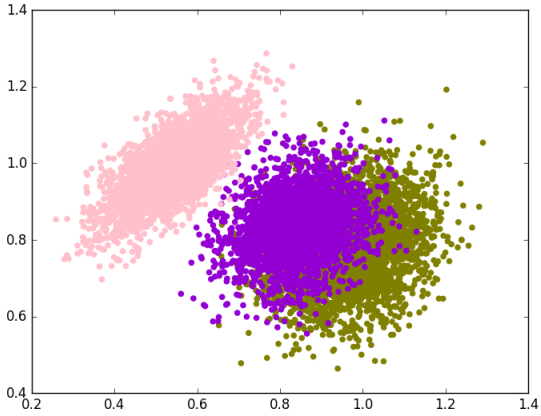
Parámetros 2

$\pi = 0.48$

$\mu = [\,0.6898, 0.7135]$

$\Sigma = \begin{bmatrix} 0.00884 & 0.006304 \\ 0.006304 & 0.008689 \end{bmatrix}$

Figura 5.1: Representación en dos dimensiones; n=1000000, K=2



Parámetros 1

$\pi = 0.34$

$\mu = [\,0.97, 0.784]$

$\Sigma = \begin{bmatrix} 0.009475 & 0.0021789 \\ 0.0021789 & 0.009475 \end{bmatrix}$

Parámetros 2

$\pi = 0.24$

$\mu = [\,0.842, 0.834]$

$\Sigma = \begin{bmatrix} 0.007498 & 0.0017801 \\ 0.0017801 & 0.007498 \end{bmatrix}$

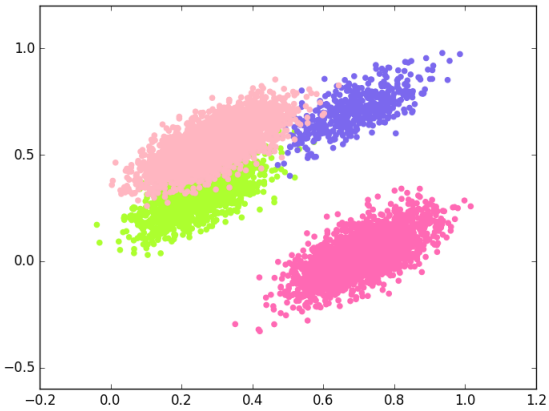
Parámetros 3

$\pi = 0.34$

$\mu = [\,0.541, 0.987]$

$\Sigma = \begin{bmatrix} 0.006715 & 0.00471 \\ 0.00471 & 0.006715 \end{bmatrix}$

Figura 5.2: Representación en dos dimensiones; n=1000000, K=3



Parámetros 1

$\pi = 0.12$

$\mu = [\,0.25168, 0.29897]$

$\Sigma = \begin{bmatrix} 0.008233 & 0.005839 \\ 0.005869 & 0.00905 \end{bmatrix}$

Parámetros 2

$\pi = 0.05$

$\mu = [\,0.68988, 0.71354]$

$\Sigma = \begin{bmatrix} 0.008847 & 0.006304 \\ 0.006304 & 0.00868 \end{bmatrix}$

Parámetros 3

$\pi = 0.30$

$\mu = [\,0.29, 0.547]$

$\Sigma = \begin{bmatrix} 0.008847 & 0.006304 \\ 0.006304 & 0.008689 \end{bmatrix}$

Parámetros 4

$\pi = 0.25$

$\mu = [\,0.71, 0.03]$

$\Sigma = \begin{bmatrix} 0.008233 & 0.005839 \\ 0.005839 & 0.009050 \end{bmatrix}$

Figura 5.3: Representación en dos dimensiones; n=1000000, K=4

Adicional a ellos se realizaron experimentos utilizando otro conjunto de datos para medir la eficacia de la implementación aquí descrita. Este conjunto de datos es el llamado *Old Faithful* el cuál se realizó a través de medir el tiempo de espera entre las erupciones y la duración de la erupción para el géiser Old Faithful en el Parque Nacional de Yellowstone, Wyoming, EE.UU [1]. Dicho conjunto de datos cuenta con 272 registros bidimensionales:

- eruptions(tipo numérico con punto flotante). Tiempo de duración de la erupción.
- waiting(tipo numérico con punto flotante). Tiempo de espera para la siguiente erupción.

Es importante mencionar que todos los conjuntos de datos fueron normalizados utilizando 5.1; como parte de un requisito que deben cumplir los conjuntos de datos que procesa el algoritmo de este trabajo.

$$Z = \frac{x - min}{max - min} \tag{5.1}$$

Dónde x es el dato a normalizar, min es el valor mínimo que x puede tomar así como, max es el valor máximo que x puede tomar en el dominio dado.

[1] Tomado de: "http://www.stat.cmu.edu/ larry/all-of-statistics/=data/faithful.dat"

5.2. Resultados

Los conjuntos de datos mencionados anteriormente se utilizaron como entradas para la ejecución del programa que se desarrolló durante este trabajo, en la versión paralela como serial.

Es importante hacer mención que los resultados descritos en esta sección se obtuvieron utilizando un clúster con arquitectura específica, es decir se asegura que los resultados puedan ser replicados siempre y cuando se ejecute el programa en una arquitectura con características similares a las que se presentan a continuación.

Se utilizó el clúster del Doctor Ricardo Barrón Fernández, quién nos facilitó el acceso mediante una cuenta, dicha infraestructura cuenta con las siguientes características:

- Sistema operativo: CentOS 6.5
- Arquitectura: x86, 32bits
- CPU(s): 4
- Thread por nucleo: 2
- Nucleo por socket: 2
- Socket(s): 1
- Dos tarjeta gráficas con las características que se presentan en las tablas 5.1 y 5.2.

Nombre	Tesla C2075
Capacidad de cómputo	2.0
Velocidad de reloj	1147000
Superposición de copia del dispositivo	Enabled
Kernel timeout	Enabled
Total memoria global	5636292608
Total memoria constantes	65536
Max memoria pitch	2147483647
Alineación de texturas	512
Número de Multiprocesadores	14
Memoria compartida por mp	49152
Registros por mp	32768
Threads en warp	32
Max threads por bloque	1024
Max thread dimensiones	(1024, 1024, 64)
Max grid dimensiones	(65535, 65535, 65535)

Tabla 5.1: Características de tarjeta gráfica Tesla C2075

Nombre	Quadro 4000
Capacidad de cómputo	2.0
Velocidad de reloj	950000
Superposición de copia del dispositivo	Enabled
Kernel timeout	Enabled
Total memoria global	2147155968
Total memoria constantes	65536
Max memoria pitch	2147483647
Alineación de texturas	512
Número de Multiprocesadores	8
Memoria compartida por mp	49152
Registros por mp	32768
Threads en warp	32
Max threads por bloque	1024
Max thread dimensiones	(1024, 1024, 64)
Max grid dimensiones	(65535, 65535, 65535)

Tabla 5.2: Características de tarjeta gráfica Quadro 4000

Dado las características sólo se trabajó con la tarjeta Tesla C2075, pues es la recomendada para tareas de supercómputo, por lo que las versiones paralelas fueron ejecutadas con esa tarjeta gráfica.

Los resultados obtenidos a partir de utilizar como entrada los conjuntos de datos que formamos haciendo uso de la aplicación desarrollada en Python. Se realizó la medición del tiempo, en milisegundos, desde el momento en que comienza la ejecución del algoritmo hasta que se obtiene los parámetros finales, es decir para obtener dichas mediciones de tiempo no se consideran ni lectura ni escritura de archivos así como tampoco impresiones en pantalla de resultados.

Se ejecutaron las versiones serial y paralela, variando el número de $K = \{2, 3, 4, 5\}$; a partir de ello se obtuvo lo siguiente:

Número de datos	Dimensión vectores dato.			
–	2	3	4	5
100	1801.9213676	9965.9976	16754.985	19542.865
1000	25730.8664322	238463.98234	3023494.334	250349.34
10000	229539.0706062	1293465.43	2039047.4352	2423443.94
100000	3783652.2636414	4369423.89	62346724.2334	7944869.4302

Tabla 5.3: Resultados en milisegundos, obtenidos para n={100,1000,10000,100000}, con *k=2*. Versión serial.

Número de datos	Dimensión vectores dato.			
–	2	3	4	5
100	2591.272705	3245.8754	5426.96053	7215.8653
1000	17654.529297	25340.8432	347532.869065	44216.9743
10000	170666.109375	228183.12975	362141.86422	421165.985
100000	791451.5	1053658.7532	15429964.8732	1586329

Tabla 5.4: Resultados en milisegundos, obtenidos para n={100,1000,10000,100000}, con *k=2*. Versión paralela.

Número de datos	Dimensión vectores dato.			
–	2	3	4	5
100	2677.364746	3874.7342	5237.3522	6923.234875
1000	8816.414062	12345.003	16234.5934	17934.23404
10000	80236.648438	170487.2341	201237.24723	283734.13
100000	795449.0625	1294445.324	1924730.42	2018429.24

Tabla 5.5: Resultados en milisegundos, obtenidos para n={100,1000,10000,100000}, con *k=3*. Versión serial.

Número de datos	Dimensión vectores dato.			
–	2	3	4	5
Num datos	2	3	4	5
100	4125.7481575	9945.784	19344.5235	23423.4234
1000	32094.4051743	46823.324	52341.23	70244.1099
10000	1050912.4860764	1820374.32462	1990324.23	3123422.94
100000	14003870.3331947	21334502.234	30303897.323	31230972.1235

Tabla 5.6: Resultados en milisegundos, obtenidos para n={100,1000,10000,100000}, con *k=3*. Versión paralela.

Número de datos	Dimensión vectores dato.			
–	2	3	4	5
100	1973.526611	3277.3452	3824.34124	4125.330934
1000	8899.270508	12342.0934	16597.4345	17234.234
10000	171617.65625	290617.33909375	360397.078125	516265.382060938
100000	800297.125	748173.77324875	2012147.04653125	194255.2828666

Tabla 5.7: Resultados en milisegundos, obtenidos para n={100,1000,10000,100000}, con *k=4*. Versión serial.

Número de datos	Dimensión vectores dato.			
–	2	3	4	5
100	4040.4472351	29234.4345	33527.253	37234.234
1000	179221.4131355	213982.3452	288925.23501	292475.4352
10000	1286167.1216488	3197243.4501	3836524.564	5998624.235
100000	13083144.5674896	8739345.7345	22093472.21234	39024874.45

Tabla 5.8: Resultados en milisegundos, obtenidos para n={100,1000,10000,100000}, con *k=4*. Versión paralela.

Número de datos	Dimensión vectores dato.			
–	2	3	4	5
100	2090.094971	109.235	4229.20266907	376.86075
1000	16124.797852	27799.151496848	40863.4627165384	48574.6635453218
10000	181373.109375	223850.691590625	439557.730570313	544343.142541969
100000	1787640.125	2457111.3518125	4726007.97407616	16172871.888086

Tabla 5.9: Resultados en milisegundos, obtenidos para n={100,1000,10000,100000}, con *k=5*. Versión serial.

Número de datos	Dimensión vectores dato.			
–	2	3	4	5
100	10753.83	587.4894	20732.8754	2655.2985
1000	235715.5222893	278934.0534	412843.4235	530983.45345
10000	2087872.0012	2312972.243	5582392.3429	6872342.12345
100000	14733919.4636345	23122523.345	48659305.45	202345234.235

Tabla 5.10: Resultados en milisegundos, obtenidos para n={100,1000,10000,100000}, con *k=5*. Versión paralela.

Derivado de esto se presentan gráficas de comparación de tiempo como 5.4, para cada uno de los conjuntos de datos las cuales se pueden consultar en el anexo 1.

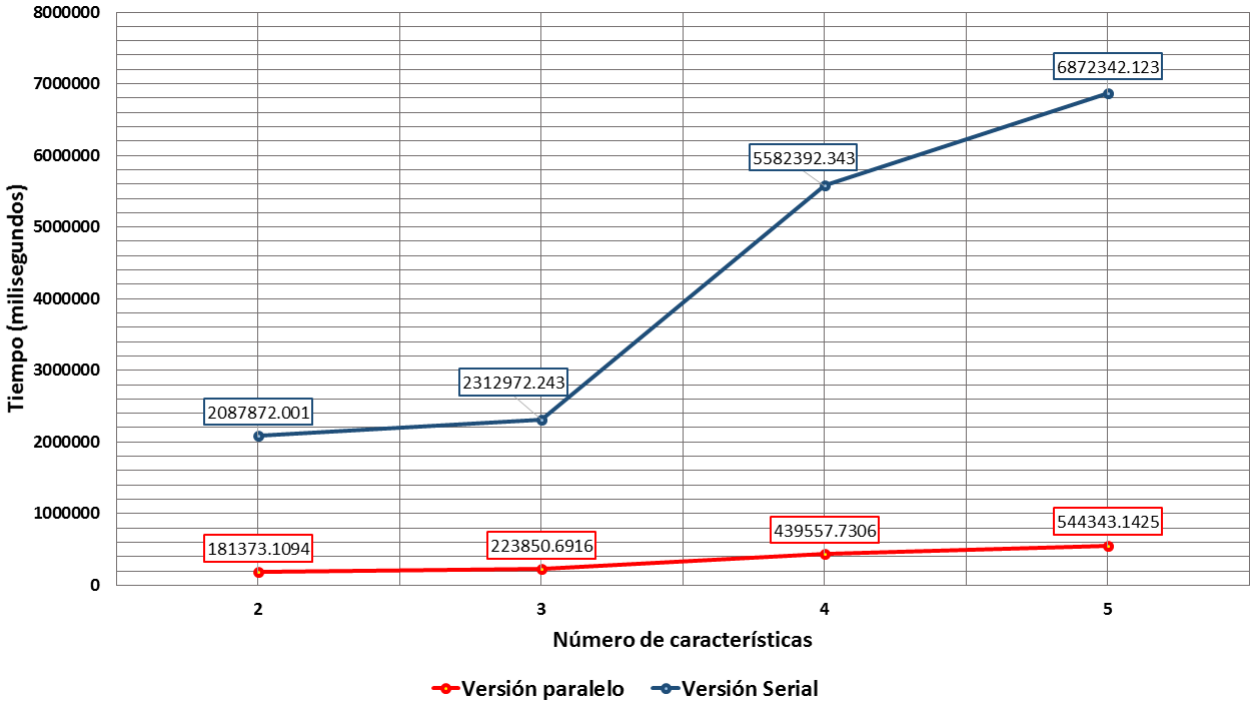


Figura 5.4: Tiempo de ejecución; n=10,000; K=5

Dichos resultados se obtuvieron con la finalidad de poder tener parámetros que nos sirvan de métricas para tener referencia sobre la mejora en el rendimiento de nuestra implementación. Las medidas que se pudieron obtener son las siguientes:

Speed-up, el cual es conocido como el factor de mejora de rendimiento y se define por 5.2, la razón entre el tiempo de ejecución de la versión paralela entre la versión serial.

$$S(n) = \frac{T(1)}{T(n)} \tag{5.2}$$

Donde n representa al número de procesadores. Se tomó el speed-up de todos los experimentos realizados, y se muestra en las siguientes tablas.

Tam. Entrada	2	3	4	5
100	0.695380831	3.070357414	3.087360763	2.708318987
1000	1.457465447	9.410262336	8.699880222	5.661837879
10000	1.344959884	5.668541015	5.630521176	5.754130263
100000	4.780649558	4.14690608	4.040626453	5.008336499

Tabla 5.11: Speed-up para K = 2

Tam. Entrada	2	3	4	5
100	1.540973513	2.566830003	3.693569338	3.383306189
1000	3.640301482	3.792896932	3.224055491	3.916761081
10000	13.09766181	10.67748171	9.890436574	11.00827363
100000	17.60498691	16.48157851	15.74448921	15.47290908

Tabla 5.12: Speed-up para K = 3

Tam. Entrada	2	3	4	5
100	2.047323412	8.920157236	8.766804763	9.025756865
1000	20.13888812	17.3376054	17.40782499	16.97060834
10000	7.494375286	11.00155779	10.645271	11.6192649
100000	16.34785901	11.68090362	10.98004853	13.79621057

Tabla 5.13: Speed-up para K = 4

Tam. Entrada	2	3	4	5
100	5.145139407	5.378215773	4.902313042	7.045834569
1000	14.61820014	10.03390529	10.10299657	10.93128423
10000	11.51147493	10.33265623	12.70002085	12.62501828
100000	8.242106036	9.410449929	10.29606927	12.51139783

Tabla 5.14: Speed-up para K = 5

El valor promedio del factor de mejora, speed-up, de la implementación fue de 8.701087 unidades.

Gene Amdahl en 1967 publicó una ley del mismo nombre, la cual dice que "la parte serial de un programa determina una cota inferior para el tiempo de ejecución, aun cuando se utilicen al máximo técnicas de paralelismo"; además asegura que la mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente. La fórmula original se escribe como:

$$T_m = T_a((1 - F_m) + \frac{F_m}{S_m}) \tag{5.3}$$

Donde:

- F_m: es la fracción del proceso original que puede mejorar.
- S_m: factor de mejora que se ha introducido (speed-up)
- T_a: tiempo de ejecución del sistema sin la mejora.
- T_m: tiempo de ejecución del sistema con la mejora

Con base en la medición del speed-up es posible calcular la eficiencia del sistema, la cual se define como:

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)} \tag{5.4}$$

Por lo que para este trabajo $n = 448$ que es el número de núcleos de procesamiento con los que cuenta la tarjeta según su manual de especificación [2]. Entonces la eficiencia del sistema es de 0.0195 unidades.

La eficiencia es una comparación del grado de speed-up conseguido frente al valor máximo. Dado que $1 \leq S(n) \leq n$, tenemos $1/n \leq E(n) \leq 1$.

[2]Consultado en: "https://www.nvidia.com/docs/IO/43395/NV-DS-Tesla-C2075.pdf"

La eficiencia más baja $E(n) \to 0$ corresponde al caso en que todo el programa ejecuta en un único procesador de forma serial. La eficiencia máxima $E(n) = 1$ se obtiene cuando todos los procesadores están siendo completamente utilizados durante todo el periodo de ejecución.

Adicional a esos experimentos, para medir la eficacia de los resultados que la implementación obtiene, se utiliza el conjunto de datos *Old Faithful*, las siguientes gráficas muestran el proceso del algoritmo en distintas iteraciones.

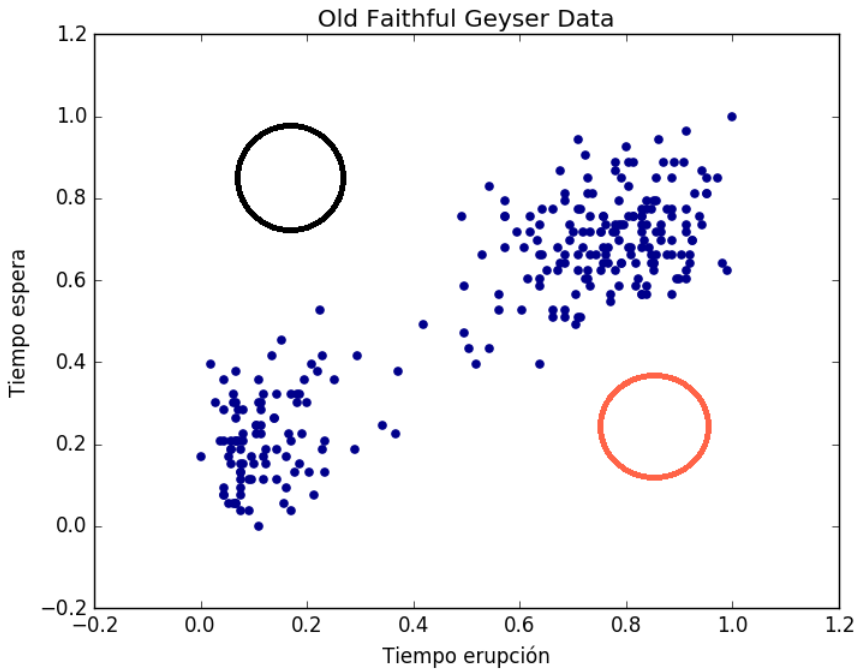


Figura 5.5: Iteración 0

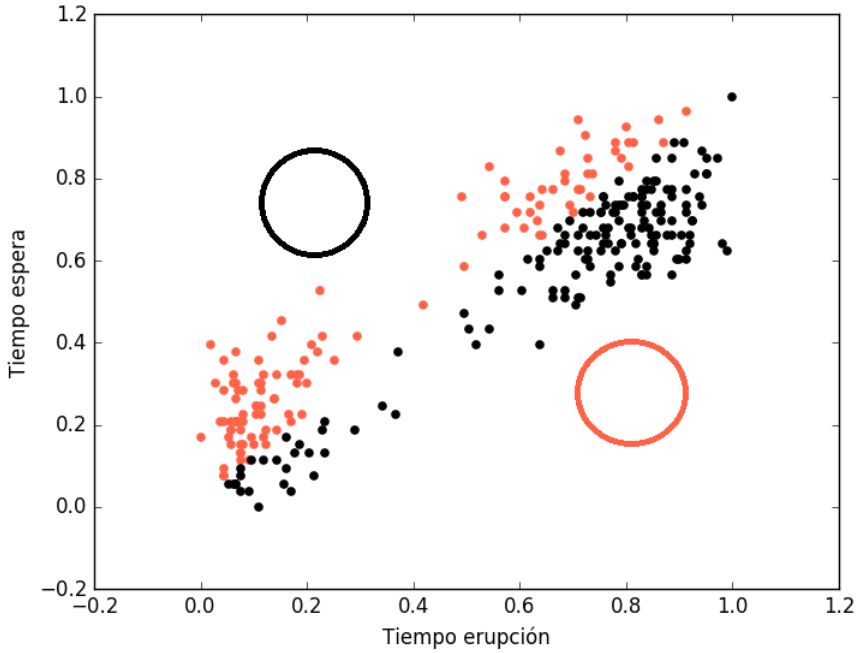


Figura 5.6: Iteración 1

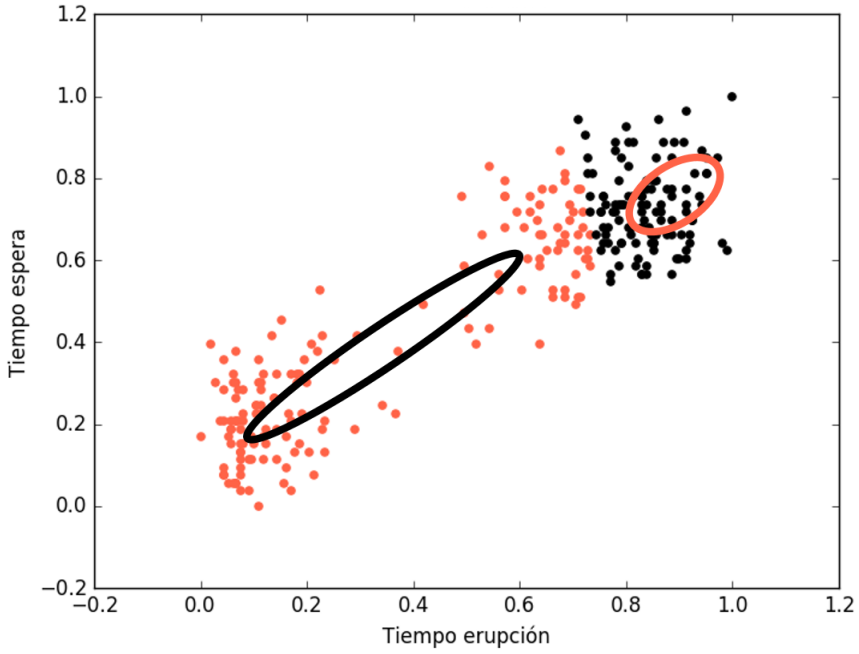


Figura 5.7: Iteración 4

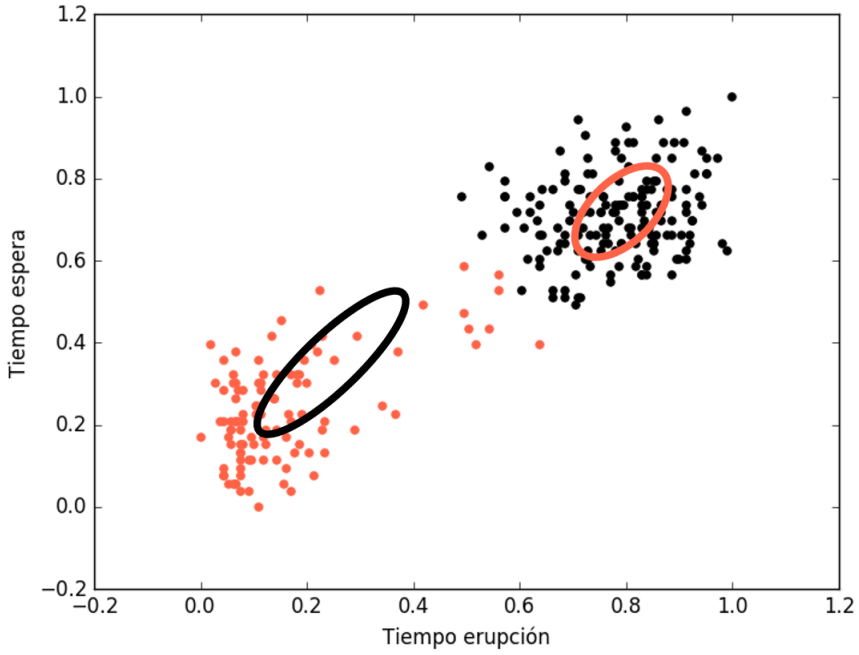


Figura 5.8: Iteración 7

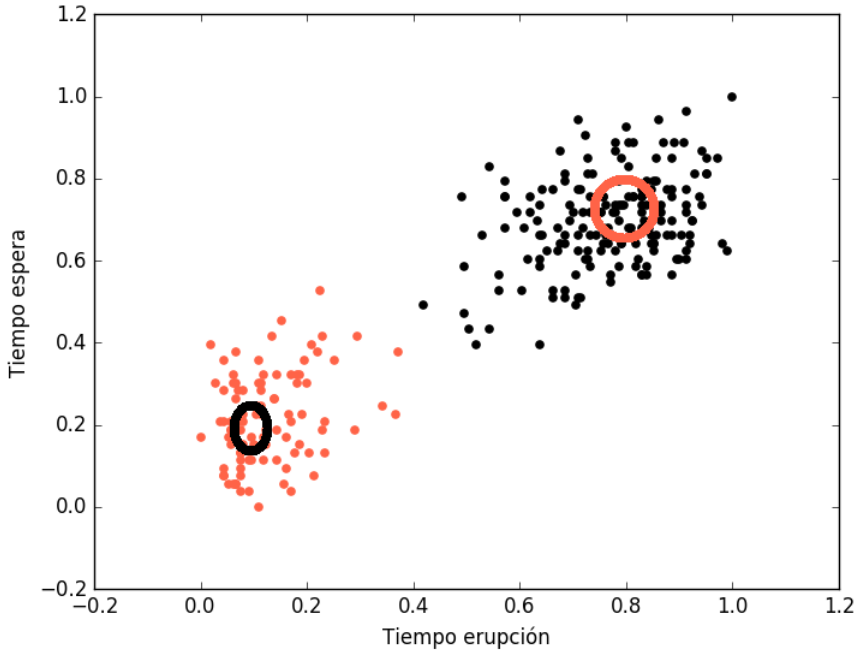


Figura 5.9: Iteración 10

Con el fin de tener parámetros de comparación se procesaron los mismos datos utilizando distintas librerías y los resultados se pueden observar en 5.15.

Implementación	Parámetros 1	Parámetros 2
Scikit learn Python	$\pi = 0{,}46029$ $\mu = \begin{pmatrix} 0{,}1281 & 0{,}2196 \end{pmatrix}$ $\Sigma = \begin{pmatrix} 0{,}0066 & 0{,}0032 \\ 0{,}0032 & 0{,}0066 \end{pmatrix}$	$\pi = 0{,}5397$ $\mu = \begin{pmatrix} 0{,}7710 & 0{,}6991 \end{pmatrix}$ $\Sigma = \begin{pmatrix} 0{,}0121 & 0{,}0045 \\ 0{,}0045 & 0{,}0121 \end{pmatrix}$
Gmdistribution class MathWorks	$\pi = 0{,}47231$ $\mu = \begin{pmatrix} 0{,}19723 & 0{,}25234 \end{pmatrix}$ $\Sigma = \begin{pmatrix} 0{,}00672 & 0{,}00321 \\ 0{,}00321 & 0{,}00671 \end{pmatrix}$	$\pi = 0{,}59886$ $\mu = \begin{pmatrix} 0{,}7701 & 0{,}69911 \end{pmatrix}$ $\Sigma = \begin{pmatrix} 0{,}01323 & 0{,}00561 \\ 0{,}00561 & 0{,}01323 \end{pmatrix}$
Mixtools R	$\pi = 0{,}460292$ $\mu = \begin{pmatrix} 0{,}1280 & 0{,}2196 \end{pmatrix}$ $\Sigma = \begin{pmatrix} 0{,}00667 & 0{,}003192 \\ 0{,}003192 & 0{,}00662 \end{pmatrix}$	$\pi = 0{,}53978$ $\mu = \begin{pmatrix} 0{,}77094 & 0{,}69909 \end{pmatrix}$ $\Sigma = \begin{pmatrix} 0{,}0122 & 0{,}004529 \\ 0{,}004529 & 0{,}0122 \end{pmatrix}$
Versión serial propuesta	$\pi = 0{,}36028$ $\mu = \begin{pmatrix} 0{,}128180 & 0{,}219676 \end{pmatrix}$ $\Sigma = \begin{pmatrix} 0{,}006566 & 0{,}00321009 \\ 0{,}0032109 & 0{,}0065066 \end{pmatrix}$	$\pi = 0{,}63970$ $\mu = \begin{pmatrix} 0{,}770954 & 0{,}69908914 \end{pmatrix}$ $\Sigma = \begin{pmatrix} 0{,}0120712 & 0{,}004529 \\ 0{,}004529 & 0{,}0012041 \end{pmatrix}$
Versión para-lela propuesta	$\pi = 0{,}46029412$ $\mu = \begin{pmatrix} 0{,}128180 & 0{,}219676 \end{pmatrix}$ $\Sigma = \begin{pmatrix} 0{,}006567 & 0{,}0032009 \\ 0{,}0032009 & 0{,}006560 \end{pmatrix}$	$\pi = 0{,}53970588$ $\mu = \begin{pmatrix} 0{,}770954 & 0{,}699089 \end{pmatrix}$ $\Sigma = \begin{pmatrix} 0{,}012072 & 0{,}004529 \\ 0{,}004529 & 0{,}012542 \end{pmatrix}$

Tabla 5.15: Resultados del algoritmo Mezclas Gaussianas con distintas aplicaciones.

Capítulo 6

Conclusiones

- El rendimiento fue mejorado con una razón promedio 8.7010; (es el promedio del speed-up) y con una eficiencia de 0.019422 unidades por procesador.
- Debido a la naturaleza de las operaciones que se manejan en el algoritmo Mezclas Gaussianas, es recomendable implementarlo en un modelo SIMD puesto que cada dato de entrada debe, en algún momento de la ejecución, ejecutar el cálculo de parámetros específicos.
- Aunque CUDA es una plataforma recomendada para este tipo de algoritmos, pues maneja el modelo SIMD, existen algunas desventajas aún sobre el manejo de la memoria compartida, sobre todo por el tamaño de esta.
- Aunque existen APIs que permiten el manejo de CUDA desde otros lenguajes de programación, es recomendable utilizar la versión extendida de C/C++.
- Es importante tener en cuenta que al utilizar C el tratamiento previo de los datos resulta ser compleja, por lo que es mejor darle tratamiento de forma separada y solo utilizar CUDA para el algoritmo principal.
- Se realizó una implementación en paralelo del algoritmo EM de Mezcla de Gaussianas.
- Se realizó una implementación en paralelo del algoritmo K-means.

Capítulo 7

Trabajo futuro

A partir de la investigación realizada para este trabajo fue posible conocer que en el estado del arte se habla de combinar las técnicas de clústeres como MapReduce, con el uso de unidades de procesamiento gráfico, con el fin de explotar las ventajas de cada uno. Sería interesante esta propuesta, pretende aminorar el problema con el ancho de banda lo cuál es posible por el sistema de archivos DFS sobre el cuál se basa MapReduce y aprovechar el poder de procesamiento de los GPUs.

Además aún queda pendiente realizar pruebas con tipos de algoritmos de aprendizaje máquina distintos Mixture of Gaussians, ya que aunque existen herramientas que ofrecen cómputo paralelo para redes neuronales, por ejemplo, sería interesante tratar de mojarar las implementaciones o ventajas que ofrecen dichas herramientas.

Uno de los trabajos a futuro más importantes es unir la imlementación obtenida por este trabajo, junto al trabajo titulado *Compresión de datos para aprendizaje de máquina* del compañero Bernardo Aurelio Gonzalez Torres, de forma que las salidas obtenidas por esta implementación sirvan como entradas a su implementación y poder obtener resultados con una precisión mayor.

Capítulo 8

Anexos

8.1. Anexo 1. Tiempo de ejecución.

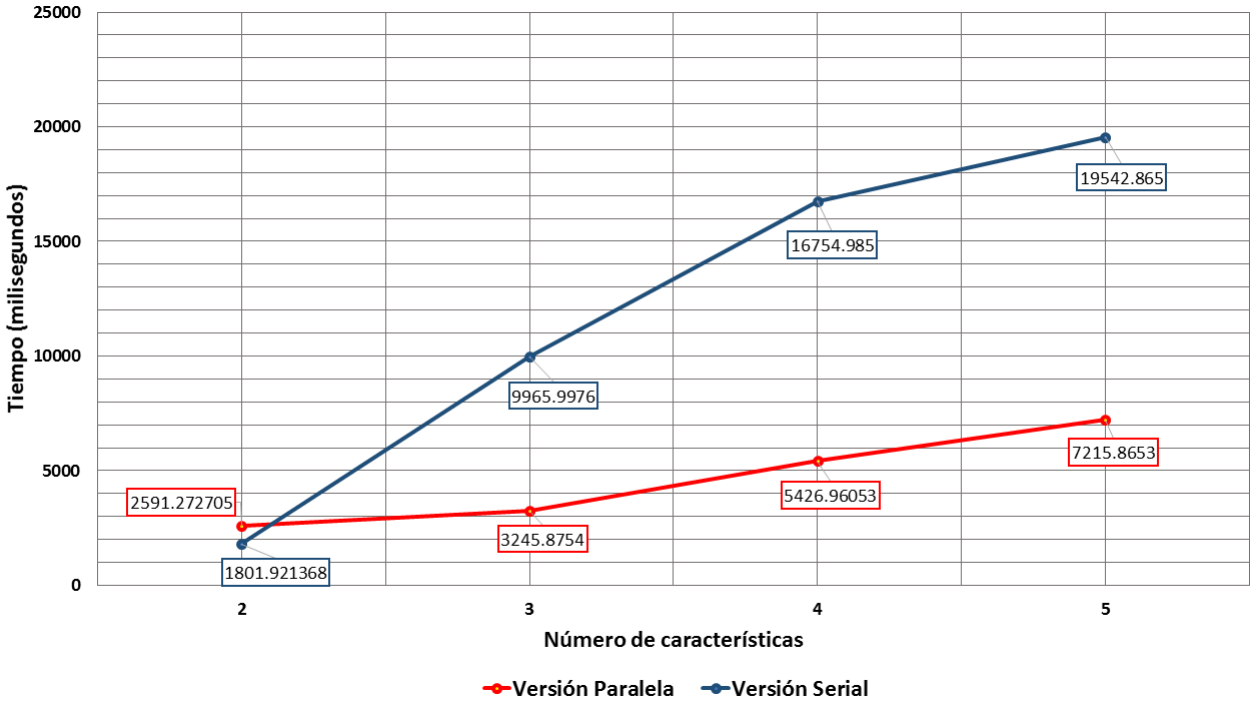


Figura 8.1: Tiempo de ejecución; n=100; K=2

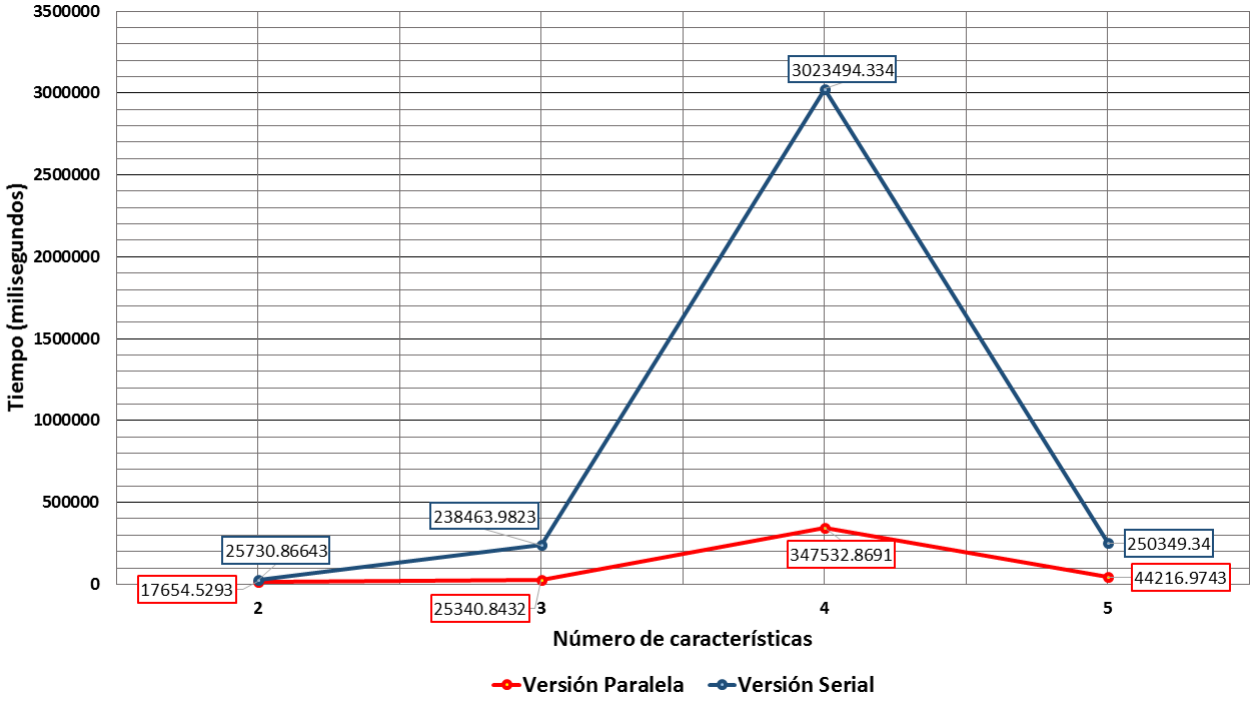


Figura 8.2: Tiempo de ejecución; n=1,000; K=2

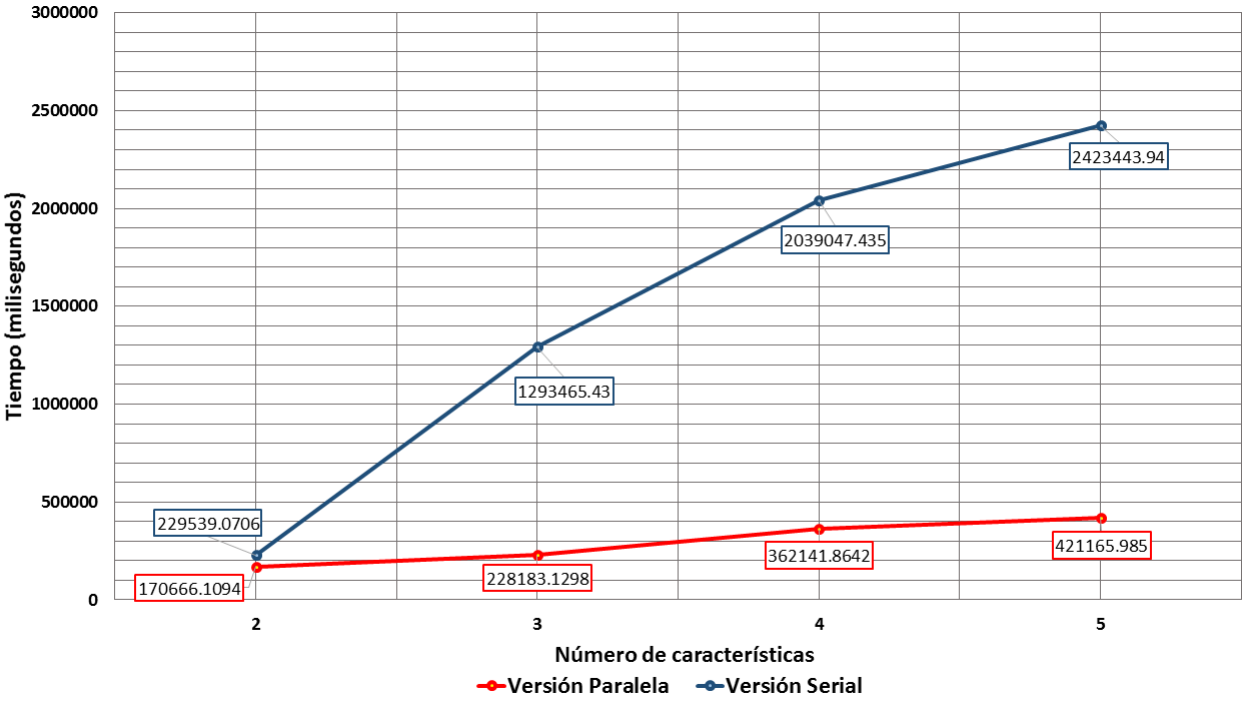


Figura 8.3: Tiempo de ejecución; n=10,000; K=2

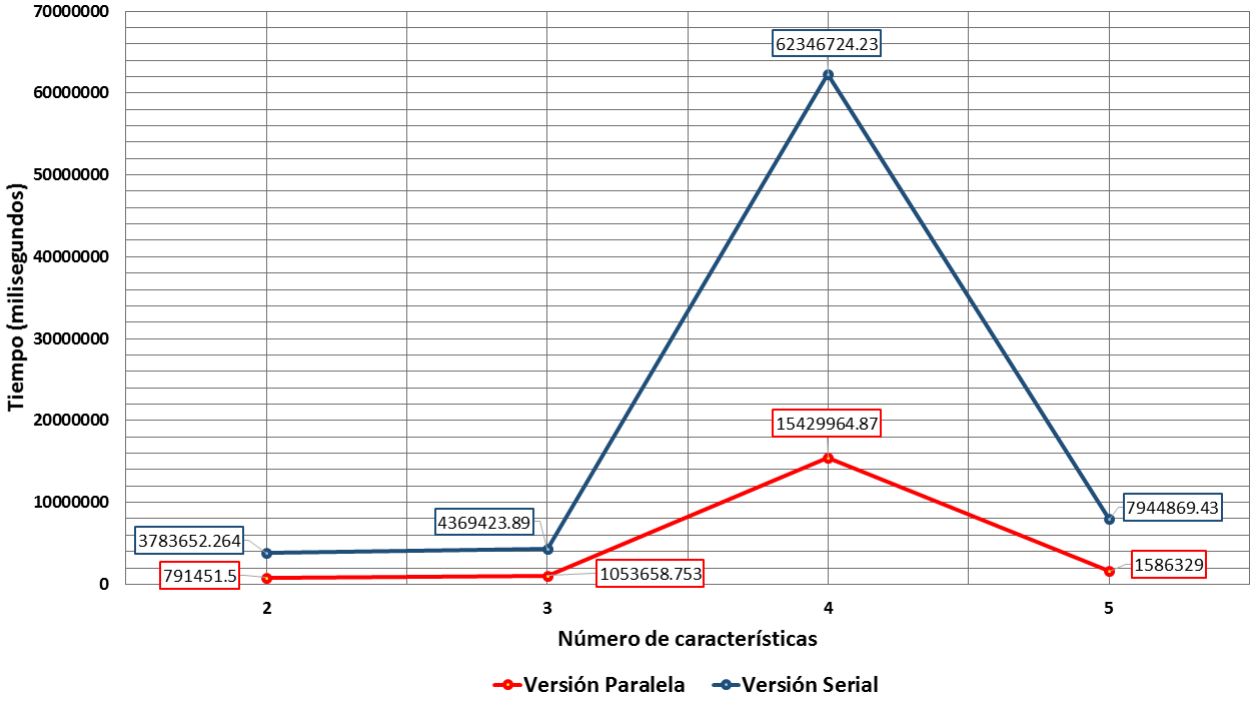


Figura 8.4: Tiempo de ejecución; n=100,000; K=2

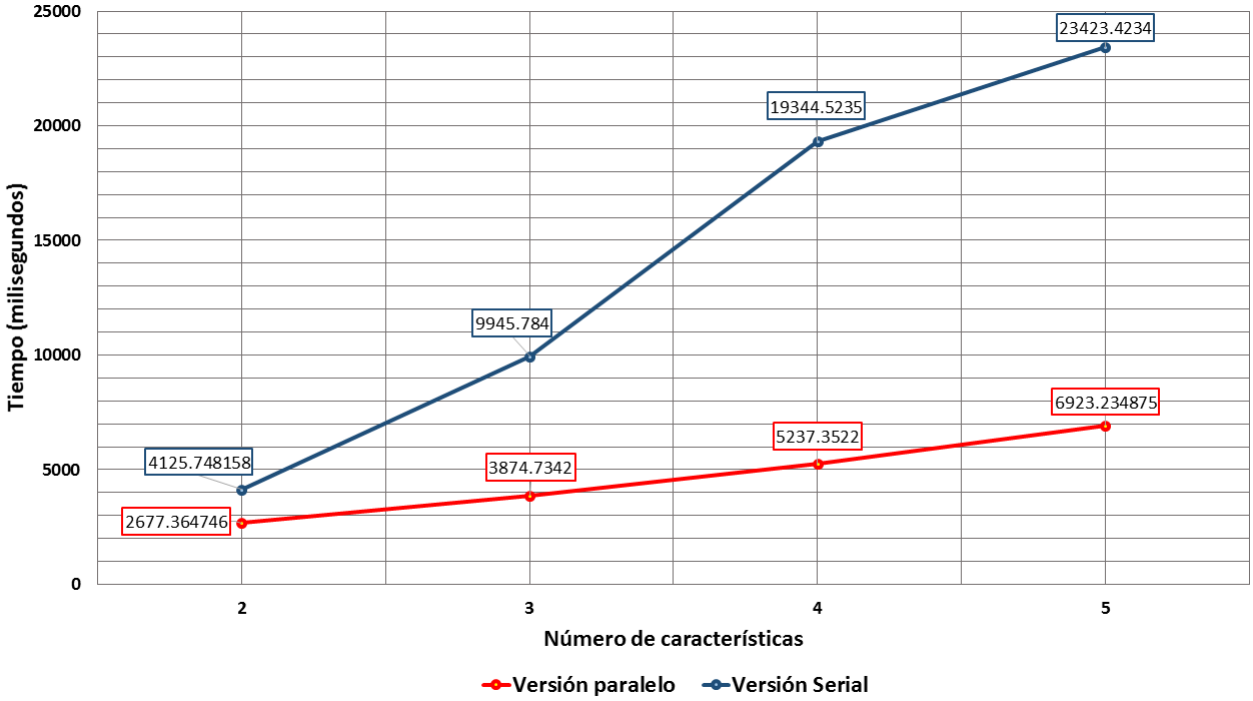


Figura 8.5: Tiempo de ejecución; n=100; K=3

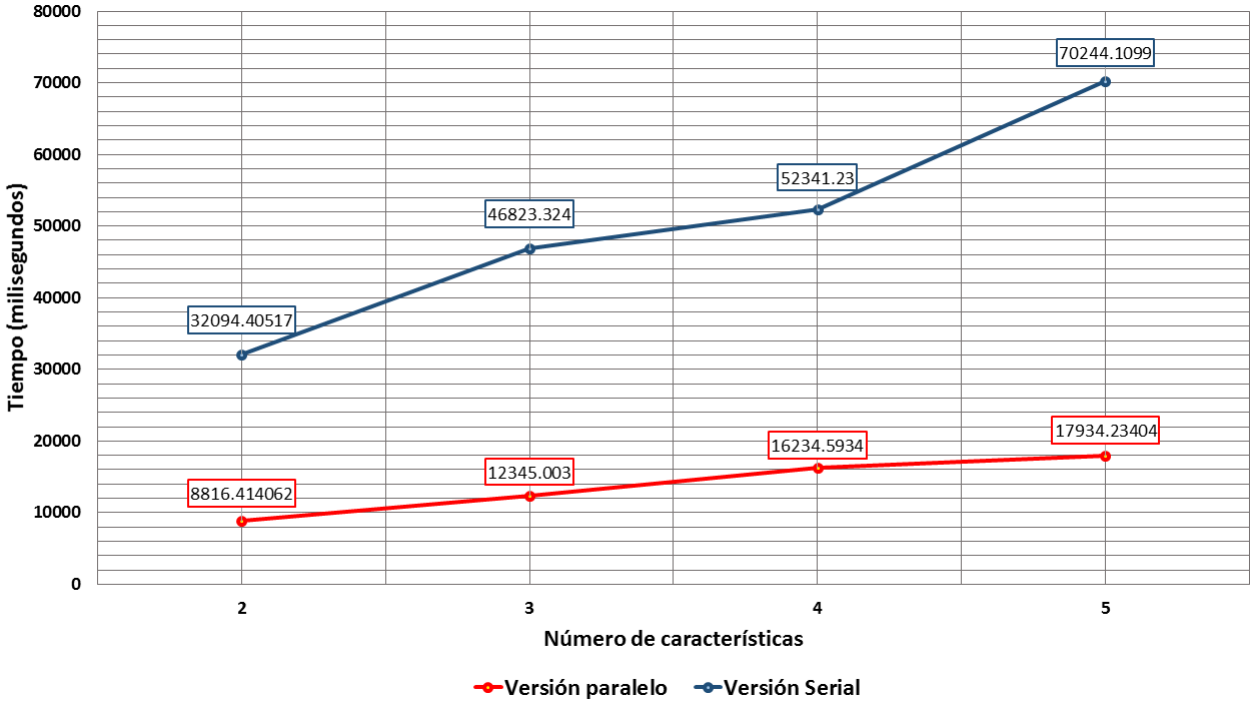


Figura 8.6: Tiempo de ejecución; n=1,000; K=3

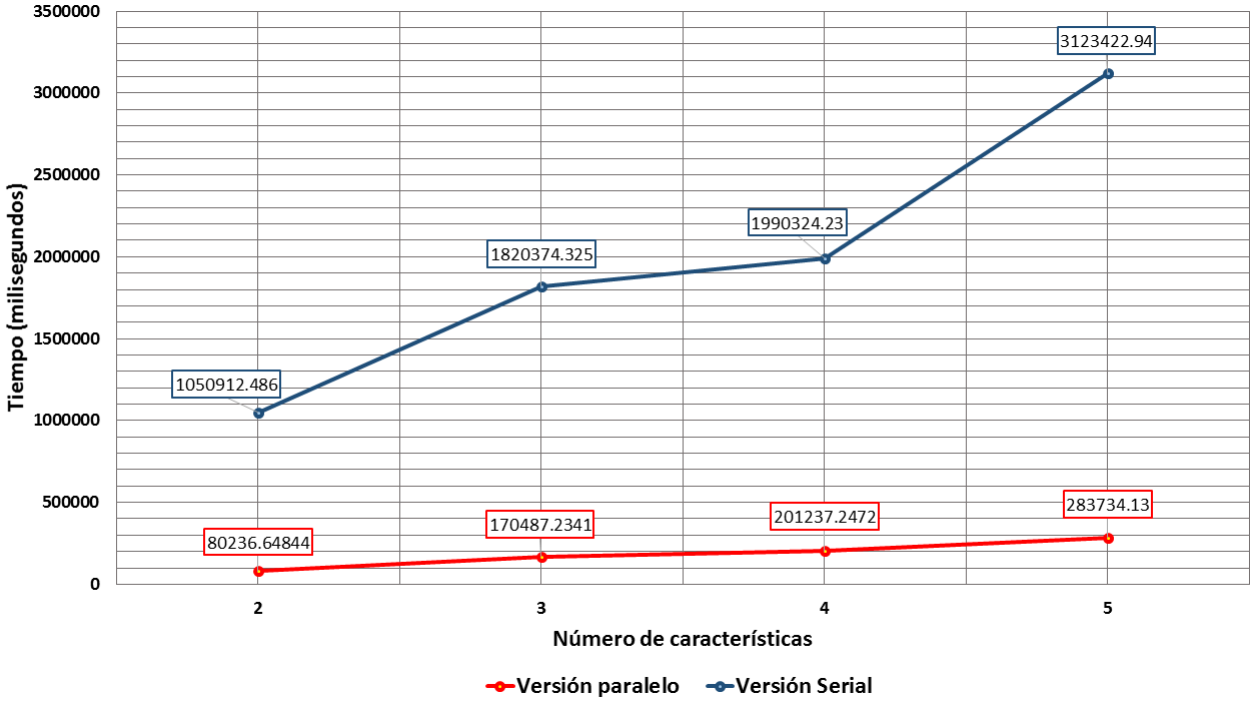


Figura 8.7: Tiempo de ejecución; n=10,000; K=3

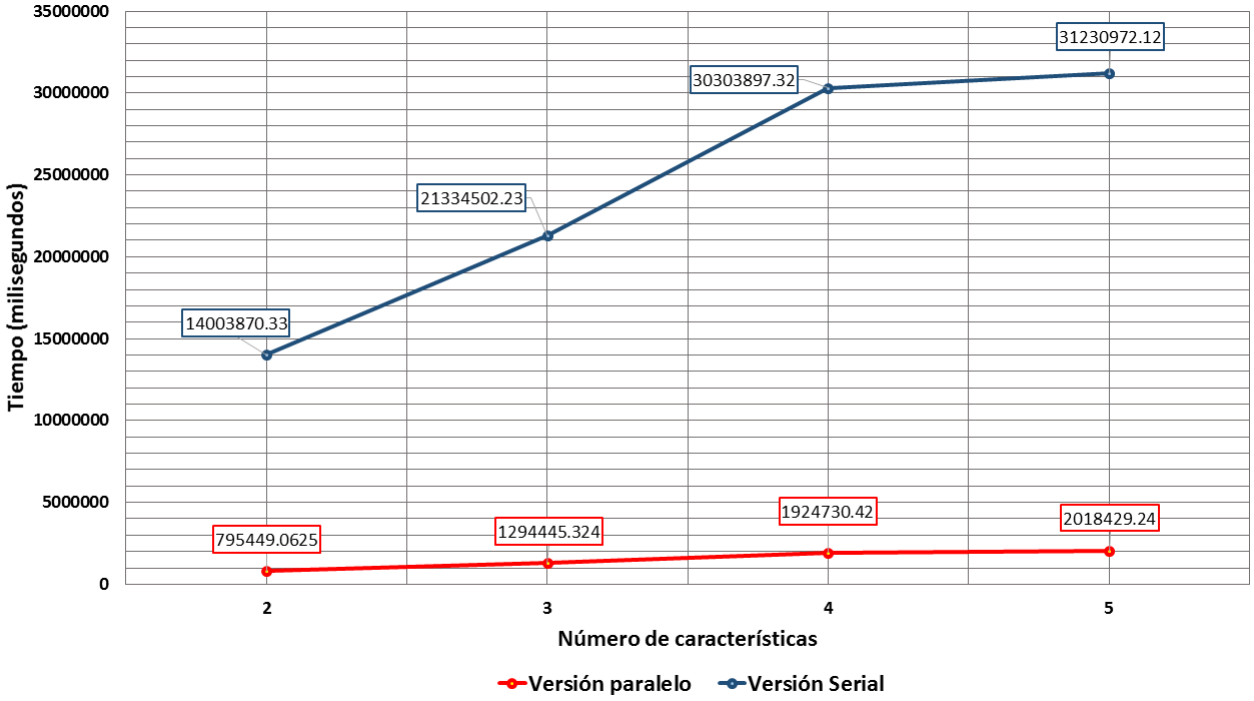


Figura 8.8: Tiempo de ejecución; n=100,000; K=3

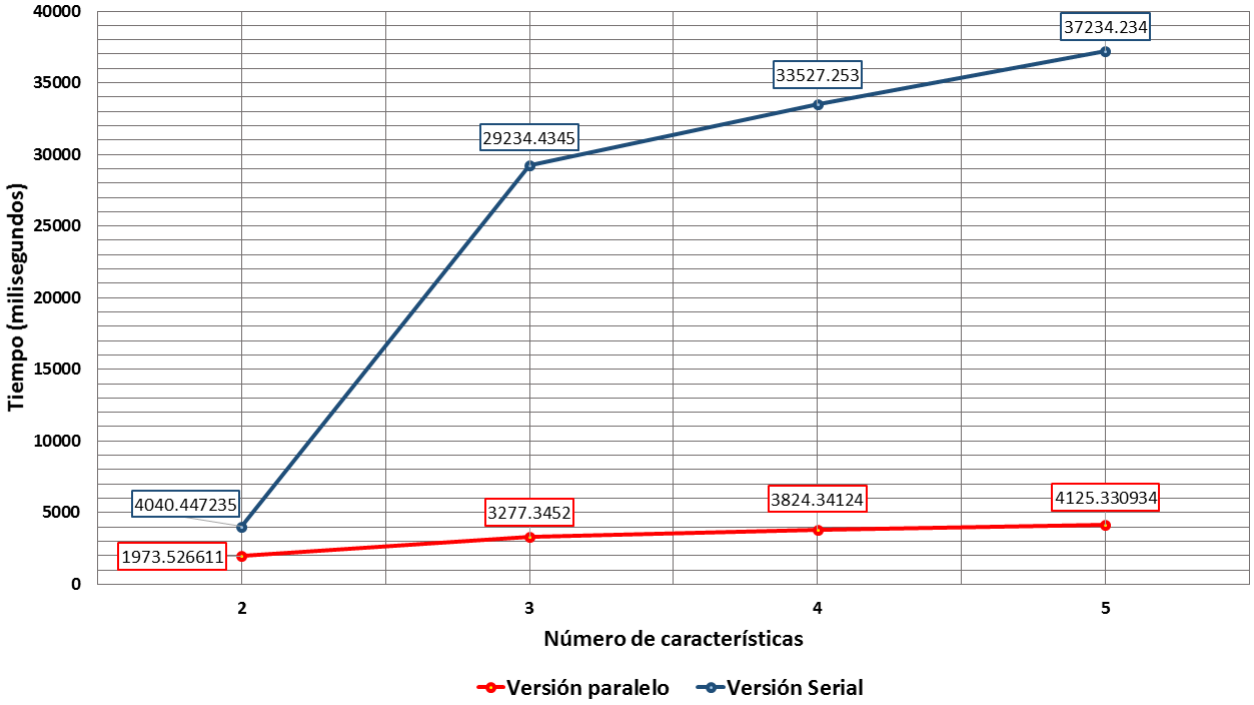


Figura 8.9: Tiempo de ejecución; n=100; K=4

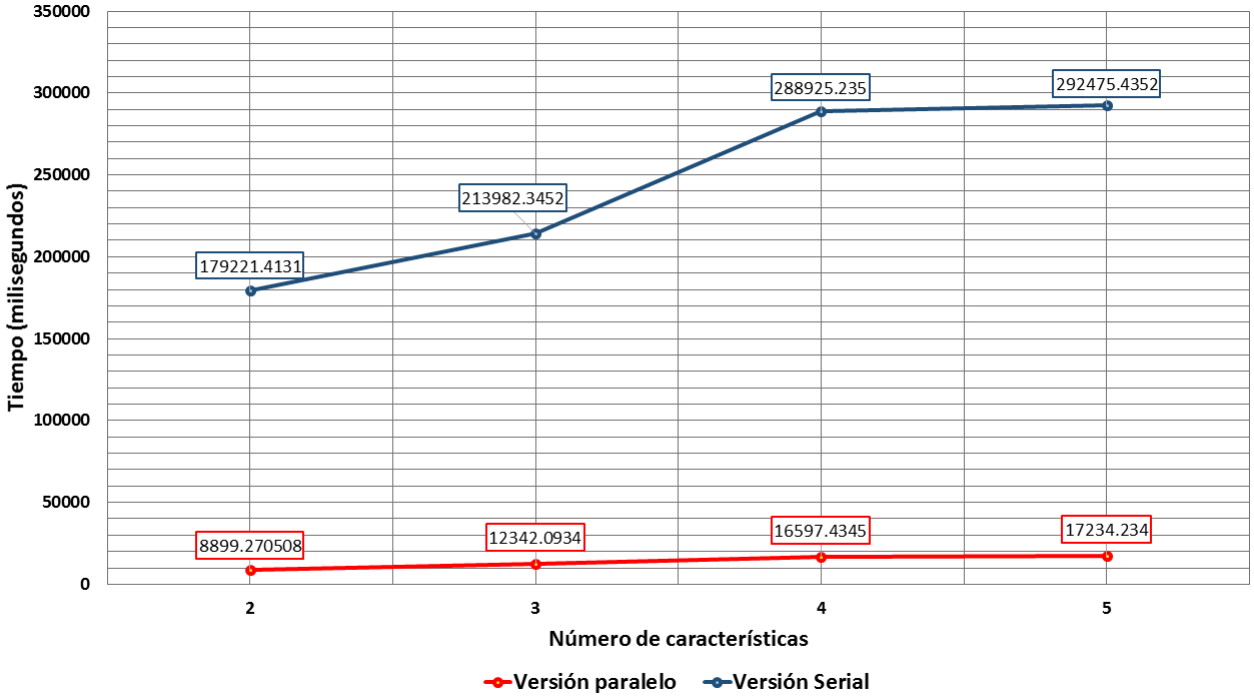


Figura 8.10: Tiempo de ejecución; n=1,000; K=4

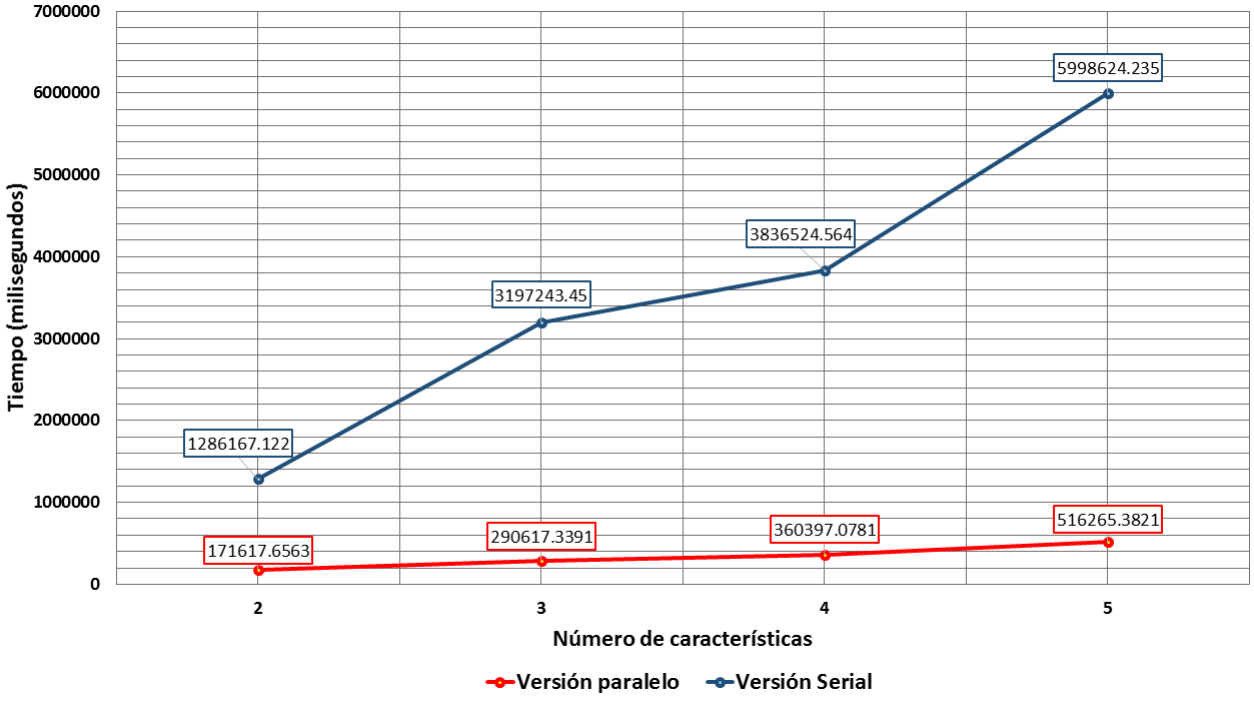


Figura 8.11: Tiempo de ejecución; n=10,000; K=4

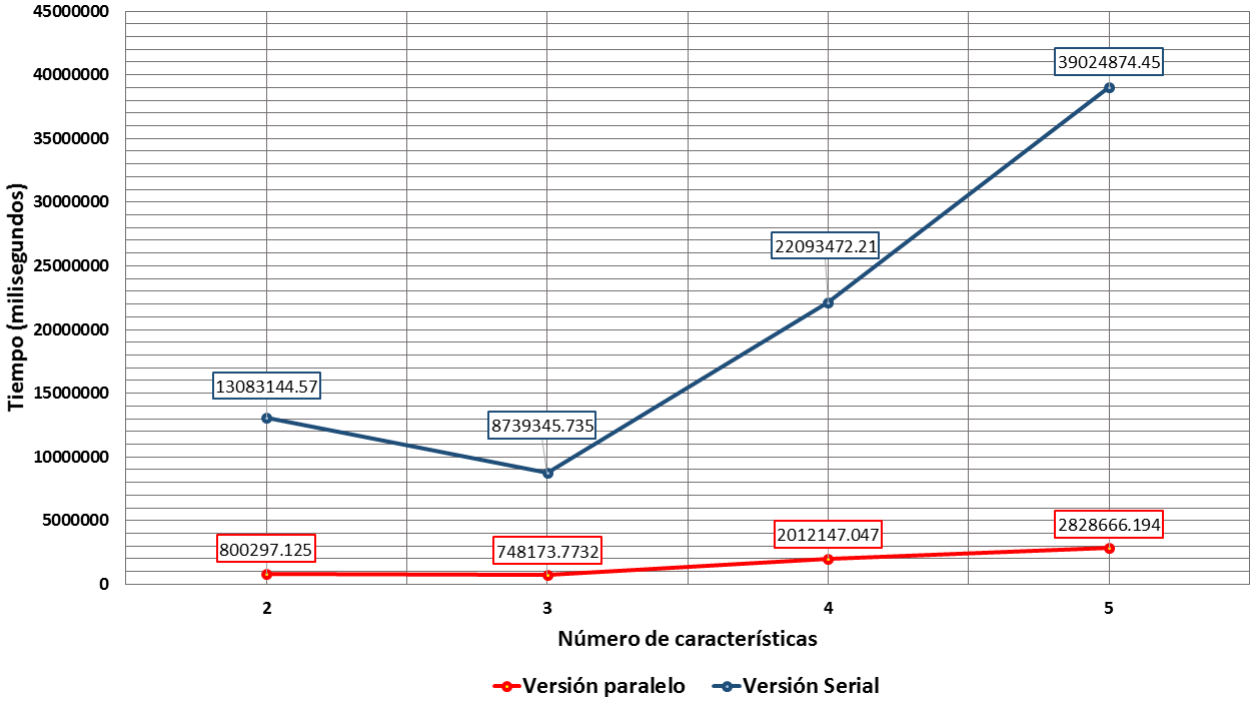


Figura 8.12: Tiempo de ejecución; n=100,000; K=4

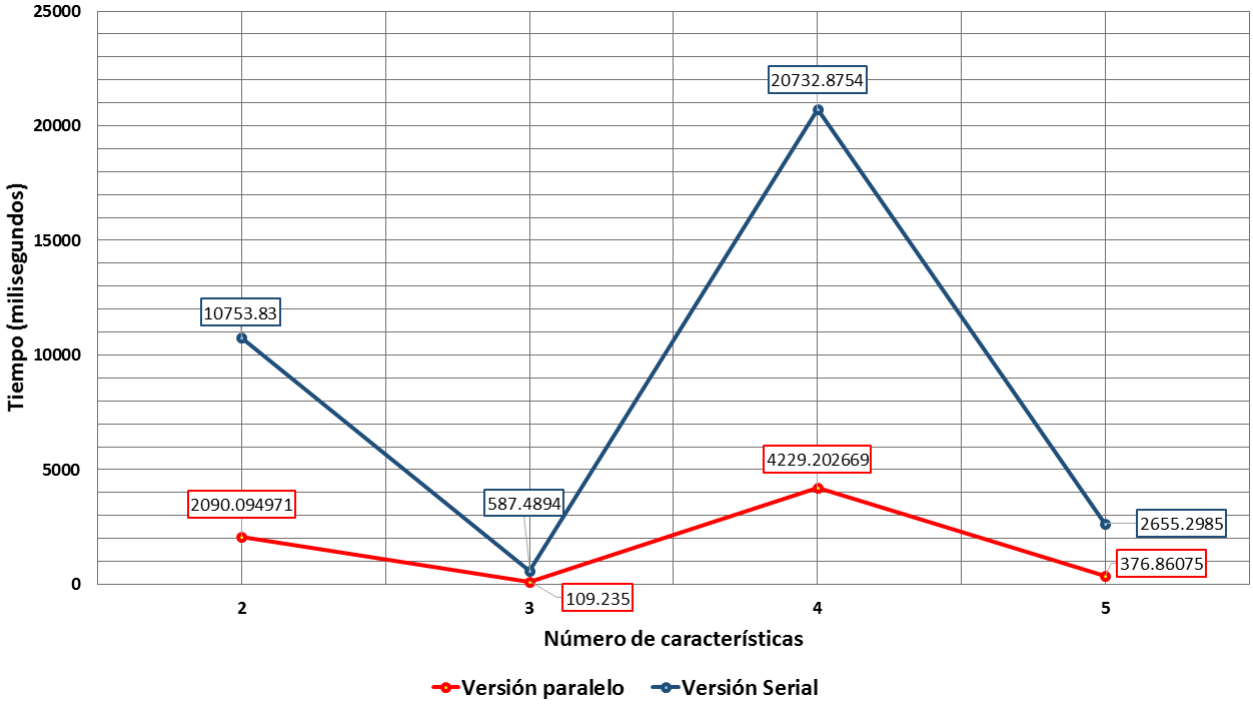


Figura 8.13: Tiempo de ejecución; n=100; K=5

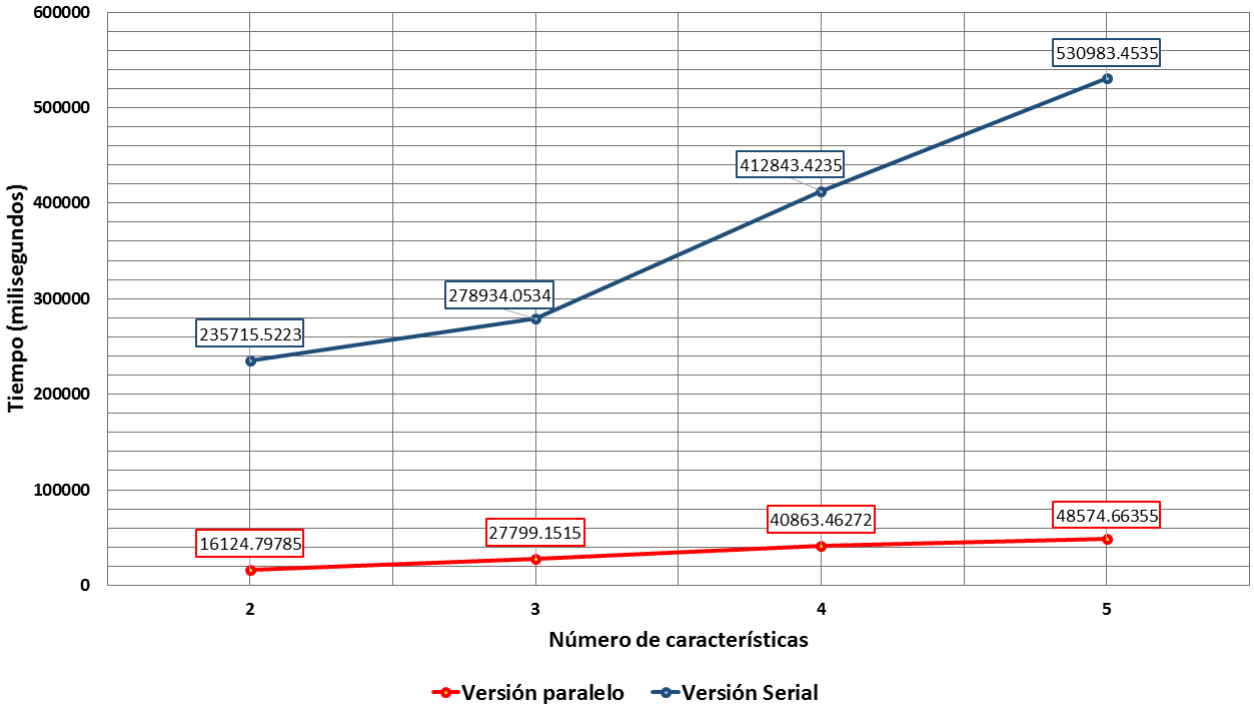


Figura 8.14: Tiempo de ejecución; n=1,000; K=5

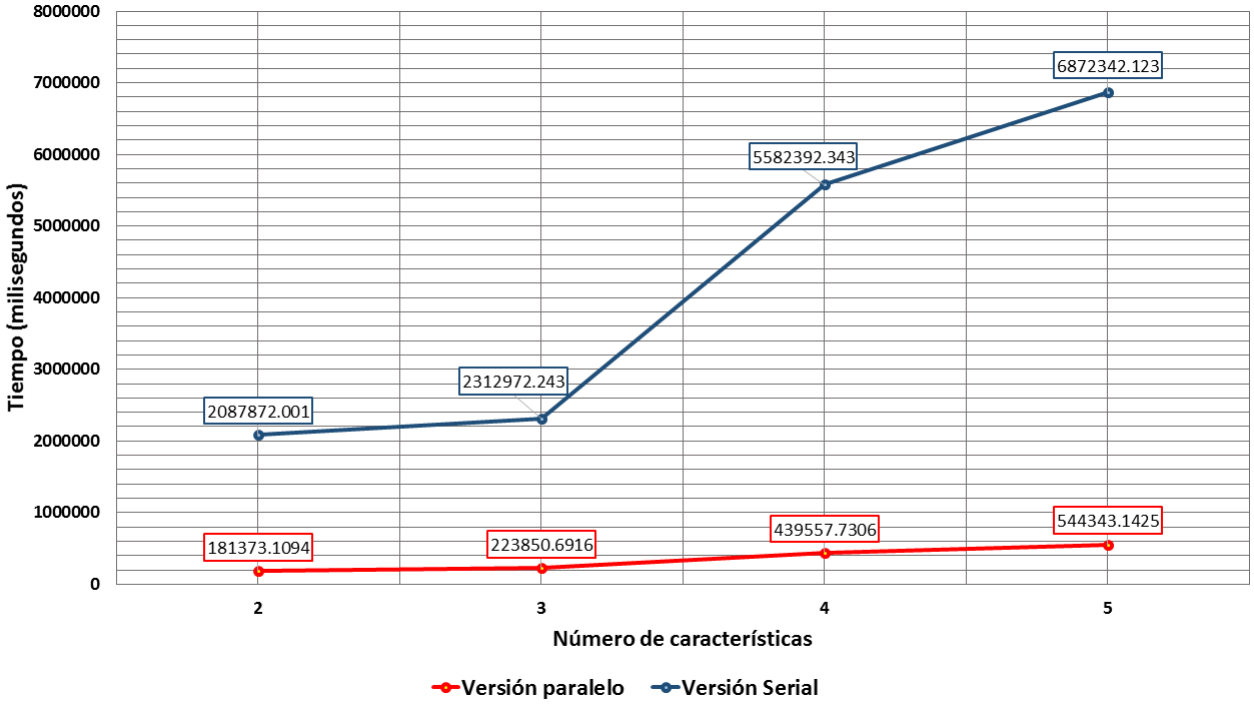


Figura 8.15: Tiempo de ejecución; n=10,000; K=5

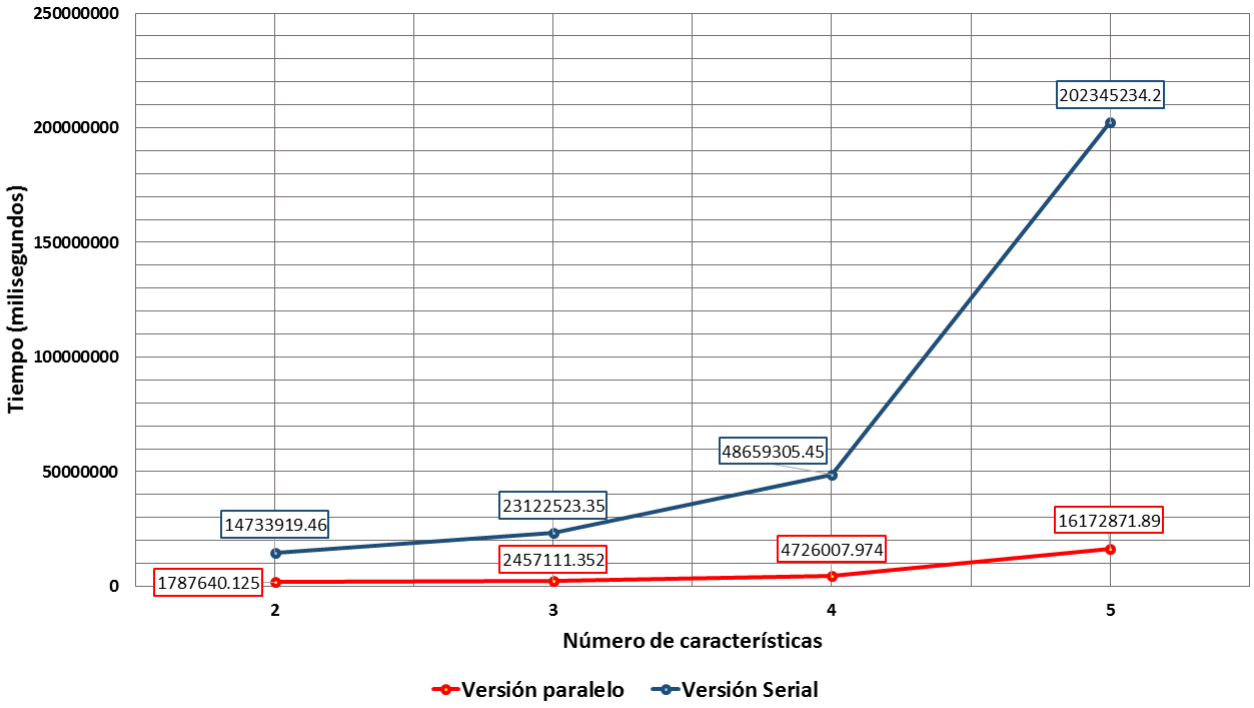


Figura 8.16: Tiempo de ejecución; n=100,000; K=5

Bibliografía

[1] Yaser S Abu-Mostafa, Malik Magdon-Ismail y Hsuan-Tien Lin. *Learning from data*. Vol. 4. AMLBook Singapore, 2012.

[2] Sanghamitra Bandyopadhyay y Sriparna Saha. *Unsupervised classification: similarity measures, classical and metaheuristic approaches, and applications*. Springer Science & Business Media, 2012.

[3] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[4] Jeff Bolz y col. "Sparse matrix solvers on the GPU: conjugate gradients and multigrid". En: *ACM Transactions on Graphics (TOG)*. Vol. 22. 3. ACM. 2003, págs. 917-924.

[5] Feng Cao, Anthony KH Tung y Aoying Zhou. "Scalable clustering using graphics processors". En: *International Conference on Web-Age Information Management*. Springer. 2006, págs. 372-384.

[6] AUSTIN Carpenter. "CUSVM: A CUDA implementation of support vector classification and regression". En: *patternsonscreen. net/cuSVMDesc. pdf* (2009).

[7] Bryan Catanzaro, Narayanan Sundaram y Kurt Keutzer. "A map reduce framework for programming graphics processors". En: *Workshop on Software Tools for MultiCore Systems*. 2008.

[8] Bryan Catanzaro, Narayanan Sundaram y Kurt Keutzer. "Fast support vector machine training and classification on graphics processors". En: *Proceedings of the 25th international conference on Machine learning*. ACM. 2008, págs. 104-111.

[9] Joseph M Cavanagh, Thomas E Potok y Xiaohui Cui. "Parallel latent semantic analysis using a graphics processing unit". En: *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*. ACM. 2009, págs. 2505-2510.

[10] Jesse St Charles y col. "Flocking-based document clustering on the graphics processing unit". En: *Nature Inspired Cooperative Strategies for Optimization (NICSO 2007)*. Springer, 2008, págs. 27-37.

[11] Shuai Che y col. "A performance study of general-purpose applications on graphics processors using CUDA". En: *Journal of parallel and distributed computing* 68.10 (2008), págs. 1370-1380.

[12] Cheng Chu y col. "Map-reduce for machine learning on multicore". En: *Advances in neural information processing systems* 19 (2007), pág. 281.

[13] Xiaohui Cui, Jesse St Charles y Thomas Potok. "GPU enhanced parallel computing for large scale data clustering". En: *Future Generation Computer Systems* 29.7 (2013), págs. 1736-1741.

[14] Luke Dicken y John Levine. "Applying clustering techniques to reduce complexity in automated planning domains". En: *Intelligent Data Engineering and Automated Learning–IDEAL 2010*. Springer, 2010, págs. 186-193.

[15] Wenbin Fang y col. "Parallel data mining on graphics processors". En: *Hong Kong Univ. Sci. and Technology, Hong Kong, China, Tech. Rep. HKUST-CS08-07* (2008).

[16] Wenbin Fang y col. "Frequent itemset mining on graphics processors". En: *Proceedings of the fifth international workshop on data management on new hardware*. ACM. 2009, págs. 34-42.

[17] Michael Geary y col. "Implementing extremely randomized trees in CUDA". En: *Department of Electrical Engineering, Stanford, CA., Tech. Rep* (2009).

[18] Amol Ghoting y col. "Cache-conscious frequent pattern mining on a modern processor". En: *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment. 2005, págs. 577-588.

[19] Naga Govindaraju y col. "GPUTeraSort: high performance graphics co-processor sorting for large database management". En: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM. 2006, págs. 325-336.

[20] Sudipto Guha, S Krisnan y Suresh Venkatasubramanian. "Data visualization and mining using the gpu". En: *Tutorial at ACM SIGKDD* 5 (2005).

[21] Bingsheng He y col. "Mars: a MapReduce framework on graphics processors". En: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM. 2008, págs. 260-269.

[22] Anil K Jain. "Data clustering: 50 years beyond K-means". En: *Pattern recognition letters* 31.8 (2010), págs. 651-666.

[23] Rahul Khanna y Mariette Awad. *Efficient Learning Machines: Theories, Concepts, and Applications for Engineers and System Designers*. Apress, 2015.

[24] Jon Kleinberg y Éva Tardos. *Algorithm design*. Pearson Education India, 2006.

[25] Brett Lantz. *Machine learning with R*. Packt Publishing Ltd, 2013.

[26] Jure Leskovec, Anand Rajaraman y Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2014.

[27] Haoyuan Li y col. "Pfp: parallel fp-growth for query recommendation". En: *Proceedings of the 2008 ACM conference on Recommender systems*. ACM. 2008, págs. 107-114.

[28] Jimmy Lin y Chris Dyer. "Data-intensive text processing with MapReduce". En: *Synthesis Lectures on Human Language Technologies* 3.1 (2010), págs. 1-177.

[29] Daniel L Ly, Volodymyr Paprotski y Danny Yen. "Neural networks on gpus: Restricted boltzmann machines". En: *see http://www. eecg. toronto. edu/moshovos/CUDA08/doku. php* (2008).

[30] G Peel McLachlan. "D.(2000). Finite mixture models". En: *New York, NY: Wiley. doi* 10 (), pág. 0471721182.

[31] Geoffrey J McLachlan y Kaye E Basford. "Mixture models. Inference and applications to clustering". En: *Statistics: Textbooks and Monographs, New York: Dekker, 1988* 1 (1988).

[32] Tom M Mitchell y col. *Machine learning. WCB*. 1997.

[33] Alok Mooley, Karthik Murthy y Harshdeep Singh. "Dismarc: A distributed map reduce framework on cuda". En: *University of Texas, Austin, Tech. Rep* (2009).

[34] Andreas Mueller. "Fast sequential and parallel algorithms for association rule mining: A comparison". En: (1998).

[35] Erkki Oja. "Unsupervised learning in neural computation". En: *Theoretical computer science* 287.1 (2002), págs. 187-207.

[36] Biswanath Panda y col. "Planet: massively parallel learning of tree ensembles with mapreduce". En: *Proceedings of the VLDB Endowment* 2.2 (2009), págs. 1426-1437.

[37] Nicholas Pilkington y Heiga Zen. "An implementation of decision tree-based context clustering on graphics processing units". En: *Eleventh Annual Conference of the International Speech Communication Association*. 2010.

[38] Raghavendra D Prabhu. "GNeuron: Parallel Neural Networks with". En: *GPU", Posters, International Conference on High Performance Computing (HiPC) 2007*. Citeseer. 2006.

[39] M Schatz y Cole Trapnell. "Fast exact string matching on the GPU". En: *Center for Bioinformatics and Computational Biology* (2007).

[40] Toby Sharp. "Implementing decision trees and forests on a GPU". En: *European conference on computer vision*. Springer. 2008, págs. 595-608.

[41] Balaji Vasan Srinivasan, H Qi y Ramani Duraiswami. "GPUML: Graphical processors for speeding up kernel machines". En: *Siam Conf. on Data Mining. Workshop on High Performance Analytics-Algorithms, Implementations, and Applications*. 2010.

[42] Kurt Thearling. "Massively parallel architectures and algorithms for time series analysis". En: *Lectures in Complex Systems, Addison-Wesley* (1996).

[43] Sujatha R Upadhyaya. "Parallel approaches to machine learning—A comprehensive survey". En: *Journal of Parallel and Distributed Computing* 73.3 (2013), págs. 284-292.

[44] Michael Wehner, Lenny Oliker y John Shalf. "A real cloud computer". En: *IEEE Spectrum* 46.10 (2009), págs. 24-29.

[45] Ren Wu, Bin Zhang y Meichun Hsu. "GPU-accelerated large scale analytics". En: *IACM UCHPC* (2009).

[46] Dongkuan Xu y Yingjie Tian. "A comprehensive survey of clustering algorithms". En: *Annals of Data Science* 2.2 (2015), págs. 165-193.

[47] Mohammed J Zaki y col. "Parallel and distributed association mining: A survey". En: *IEEE concurrency* 7.4 (1999), págs. 14-25.

[48] Mario Zechner y Michael Granitzer. "Accelerating k-means on the graphics processor via cuda". En: *Intensive Applications and Services, 2009. INTENSIVE'09. First International Conference on*. IEEE. 2009, págs. 7-15.

[49] Yongpeng Zhang y col. "Large-scale multi-dimensional document clustering on GPU clusters". En: *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE. 2010, págs. 1-10.

[50] Yongpeng Zhang y col. "Data-intensive document clustering on graphics processing unit (GPU) clusters". En: *Journal of Parallel and Distributed Computing* 71.2 (2011), págs. 211-224.