TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA.

Desarrollo e Implementación de modelos paralelos de Soft Computing en CUDA

Autor
David Criado Ramón

Directores
Manuel I. Capel Tuñón
María del Carmen Pegalajar Jiménez

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, junio de 2019

Desarrollo e Implementación de modelos paralelos de Soft Computing en CUDA

David Criado Ramón

Palabras clave: palabra_clave1, palabra_clave2, palabra_clave3, ...

Resumen

Poner aquí el resumen.

Project Title: Project Subtitle

David Criado Ramón

Keywords: Keyword1, Keyword2, Keyword3, ...

Abstract

Write here the abstract in English.

Yo, **David Criado Ramón**, alumno de la titulación Grado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 26254133-R, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: David Criado Ramón

Granada a X de mes de 2019.

D. **Manuel Capel Tuñón**, Profesor del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada.

D. **María del Carmen Pegalajar Jiménez**, Profesora del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Desarrollo e Implementación de modelos paralelos de Soft Computing en CUDA***, ha sido realizado bajo su supervisión por **David Criado Ramón**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a X de mes de 201 .

Los directores:

Manuel I. Capel Tuñón **María del Carmen Pegalajar Jiménez**

Agradecimientos

A Rubén, por estar siempre apoyándome.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción y motivación.

1.1. Motivación.

La tecnología propietaria *CUDA (Computer Unified Device Architecture)* [1] de NVIDIA, presentada en junio de 2007 y aplicable tanto a la arquitectura de las tarjetas gráficas de la misma marca como al modelo de programación genérico asociado, a lo largo de la última década ha supuesto un gran cambio en las implementaciones paralelas de algoritmos y, además, es muy utilizada y popular entre la comunidad científica.

La estructura de la GPU, utilizando un mayor número de núcleos a cambio de una velocidad de reloj más baja a la que podemos encontrar en una CPU, es de especial utilidad en operaciones masivamente paralelas, pudiendo llegar a proporcionar ganancias muy superiores con respecto al uso de la CPU.

Por otro lado, los algoritmos y técnicas de *Soft Computing* se corresponde con una rama de la Inteligencia Artificial en la que no podemos calcular soluciones exactas en tiempo polinómico y/o en los que la información es incompleta, incierta o inexacta.

El propósito de este trabajo de fin de grado es la implementación en CUDA de algunos de estos modelos de *Soft Computing* y, tras evaluar varias opciones, se optó por desarrollar dos: los mapas autoorganizados de *Kohonen* [2] y los árboles de decisión [3]. Además, para evaluar el rendimiento de las implementaciones desarrolladas, usaremos conjunto de datos con un elevado número de muestras, combinando el uso de *CUDA* y el *framework* para clústers de computación distribuida *Spark* [4].

En definitiva, en este documento, explicaremos dichos modelos, analizaremos sus posibilidades de paralelización, realizaremos las implementaciones asociadas y evaluaremos los resultados obtenidos.

1.2. Estado del arte: trabajos relacionados.

Tanto la paralelización de los mapas autoorganizados de Kohonen como la de los árboles de decisión son problemas que han sido previamente estudiados para su paralelización en CUDA.

En ***Parallel High Dimensional Self Organizing Maps Using CUDA*** Codevilla, Bothelo, Filho y Gaya [5] proponen una implementación en CUDA para la formulación tradicional del mapa autoorganizado de Kohonen. En ella, proponen una versión en la que cada iteración se realiza en 3 fases. Una primera en la que con un valor p arbitrario menor que el número de hebras por bloque que indica cuantos "pasos" debe realizar una hebra para el cálculo de la distancia euclídea, una reducción para encontrar la mejor distancia y una adaptación de pesos de neuronas basada en la dimensión del problema.

En ***Parallel Batch Self-Organizing Map on Graphics Processing Unit Using CUDA*** Daneshpajouh, Delisle, Boisson, Krajecki y Zakaria [6] plantean una adaptación en CUDA para la versión iterativa de cómputo en *batchs* del mapa autoorganizado de Kohonen. En ella aprovechan las capacidades de concurrencia disponibles en los dispositivos CUDA, paralelizando parte del algoritmo y dejando la fase de adaptación de los pesos de neuronas para ser realizada en la CPU.

Con respecto a los árboles de decisión, ***CUDT: a CUDA based decision tree algorithm*** , de Lo, Chang, Sheu, Chiu y Yuan [7], será la base de la implementación que nosotros vamos a realizar y se basa en el uso de la operación de la suma prefija, suma acumulada o *scan* para resolver un cierto tipo de árboles de decisión específicos, en concreto, árboles de decisión cuyo objetivo es la clasificación de problemas con respuesta binaria.

Aparte de la aproximación por especialización presentada en el trabajo anterior, es otra alternativa, más frecuente y versátil en la variedad de problemas que puede resolver, la discretización de las variables utilizadas durante la construcción del árbol y el uso de histogramas para ello. Esto lo podemos ver en ***Implementing Streaming Parallel Decision Trees on Graphic Processing Units***, de Svantesson [8], donde el objeto principal de su trabajo es paralelizar en CUDA los cálculos asociados a los histogra-

mas utilizados en SPDT (Streaming Parallel Decision Trees) para generar un árbol.

1.3. Objetivos.

- Iniciarse, estudiar y profundizar en el desarrollo de algoritmos paralelos en *CUDA*.
- Analizar algoritmos de *Soft Computing*, evaluando las capacidades que tienen para ser paralelizados.
- Implementar los algoritmos seleccionados en *CUDA*.
- Combinar el uso de *CUDA* y *Spark* para resolver la paralelización masiva de problemas de forma eficiente.
- Utilizar conjuntos de datos de *Big Data* que sean computacionalmente exigentes para el desarrollo de las pruebas.
- Realizar una evaluación de la calidad de los resultados obtenidos.

1.4. Estructura del documento.

- En el primer capítulo, **Introducción y motivación**, hemos comentado los própositos para la realización de este trabajo y el grado de consecución de los objetivos planteados.
- En el segundo capítulo, **Modelos de Soft Computing considerados**, explicamos los fundamentos teóricos de los algoritmos de *Soft Computing* que hemos decidido paralelizar.
- En el tercer capítulo, **Implementación**, comentamos el proceso de desarrollo seguido así como explicamos las soluciones finales implementadas y comentamos algunas de las alternativas y problemas que surgieron durante la realización de las implementaciones.
- En el cuarto capítulo, **Desarrollo de pruebas y análisis de resultados**, indicamos qué pruebas se han realizado, mostramos los resultados obtenidos y analizamos en profundidad las implicaciones de los mismos.
- En el último capítulo, **Conclusiones y trabajos futuros**, finalizamos el trabajo destacando las implicaciones más importantes de los resultados obtenidos y mostramos posibles alternativas para amplíar nuestro trabajo.

Capítulo 2

Modelos de Soft Computing considerados.

2.1. Mapas autoorganizados *(Self Organizing Map)*

A principio de la década de los 80 el científico finlandés Teuvo Kohonen [2] planteó un modelo de aprendizaje automático no supervisado y competitivo basándose en el funcionamiento del estudio del córtex cerebral. El modelo planteado, denominado mapa autoorganizado, red autoorganizada o red neuronal de Kohonen, entre otros nombres similares, es una red neuronal artificial, y las principales características que la define son las siguientes:

- Es una **red neuronal artificial**. Esto quiero decir, a grandes rangos, que la estructura que genera el modelo está basada en una red de múltiples neuronas que se encuentra interconectadas entre sí.
- La red neuronal de Kohonen tiene **dos capas**. Una capa de entrada, con tantas neuronas como características tenga una muestra a ser evaluada por la red, y una capa de salida de un tamaño que decide el usuario. Habitualmente, esta capa de salida, también llamada capa competitiva o capa de Kohonen, presenta una distribución bidimensional, aunque podría perfectamente usarse cualquier otro número de dimensiones.
- Cada neurona de la capa de entrada está asociada con todas las neuronas de la capa de salida y las neuronas de la capa de salida no están interconectadas entre sí. A este tipo de red neuronal, en la que no existen ciclos, se le denomina **red neuronal prealimentada** *(feed-forward)*.
- Asociada a cada neurona de la capa de salida, tenemos un vector de

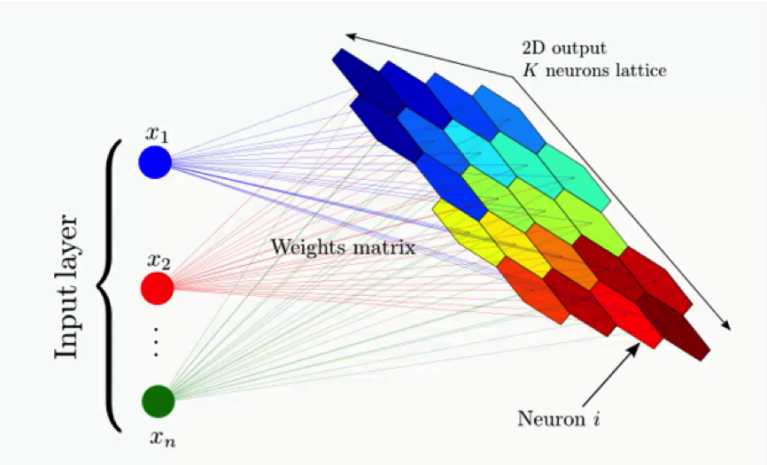


Figura 2.1: Esquema de una red neuronal de Kohonen.

pesos sinápticos obtenido a través de las conexiones con la capa de entrada que es modificado durante el proceso de aprendizaje. A dicho vector de pesos se le llama vector de referencia y representa el valor promedio de la categoría asociada a esa neurona. El conjunto de todos esos vectores de referencia es denominado *codebook*.

- Es un algoritmo **no supervisado** capaz de encontrar patrones comunes basándose en los datos de la muestra de entrada sin necesidad de que cuando una muestra entre a la red se indique a qué categoría pertenece.

- Es un modelo **competitivo**. Cuando se recibe una muestra todas las neuronas compiten por ser activadas pero sólo la mejor será activada.

2.1.1. Proceso de entrenamiento.

En primer lugar, se **inicializan los pesos** asociados a la capa de salida. Lo más habitual, es tomar dichos pesos de una distribución aleatoria. Para un correcto funcionamiento dichos pesos deben estar normalizados entre 0 y 1. En nuestro caso, hemos tomado los pesos de una distribución aleatoria uniforme en el intervalo $[0, 1)$.

El proceso de entrenamiento que se presenta a continuación **se repite hasta que se alcanza el número de iteraciones máximo**, determinado por el parámetro λ. La variable t representa cada una de las iteraciones.

Después, para cada una de las muestras, X, sacadas de la distribución de muestras, de forma aleatoria, se realizan los siguientes pasos:

1 - Se **calcula la distancia euclídea** entre la muestra X y cada una de las neuronas de la capa de salida. También se pueden utilizar otro tipo de distancias.

$$distancia(X) = ||W - X||$$

2 - Se **busca la neurona que ha obtenido una menor distancia**. Esta neurona es considerada la neurona ganadora o BMU *(Best Matching Unit)*.

$$BMU_X = argmin_{W_{i,j}}\ distancia(X) = argmin_{W_{i,j}}||W - X||$$

La función *argmin* devuelve el índice del array en el que se alcanza el valor mínimo.

3 - Se realiza un **proceso de actualización de las matrices de pesos** en base a lo obtenido anteriormente, según la siguiente fórmula.

$$W_{i,j}^{(t+1)} = W_{i,j}^{(t)} + \Delta W_{i,j}$$

La actualización depende tanto de la distancia de la muestra al vector de pesos como de otros dos parámetros: la tase de aprendizaje y una función de vecindario.

$$\Delta W_{i,j} = \eta(t)\delta_f(i,j)(X - W_{i,j})$$

La función $\delta_f(i,j)$ es la función de vecindario y, en nuestra propuesta, se calcula conforme a la siguiente función gaussiana:

$$\delta_f(i,j) = e^{-\frac{||BMU_X - (i,j)||^2}{2\sigma(t)^2}}.$$

Al tratar con una potencia con exponente negativo, un mayor valor absoluto de dicho exponente nos proporciona una valor de $\delta_f(i,j)$ menor. Por eso en el numerador se tiene en cuenta la distancia que hay entre la mejor neurona y la neurona actual. En el denominador se utiliza un parámetro de control σque nos permite controlar la distancia que estamos considerando.

Normalmente, este parámetro, durante un primer número de iteraciones previamente proporcionado, es inicializado a un valor alto σ_0 que decrece de manera exponencial conforme a otro parámetro de control τ.

Una vez ha finalizado esa primera fase (han pasado z iteraciones) se van refinando los resultados con un valor fijo mucho más bajo σ_f.

$$\sigma(t) = \begin{cases} \sigma_0 e^{-\frac{t}{\tau}} & si \;\; t < z \\ \sigma_f & si \;\; t \geq z \end{cases}$$

Para la tasa de aprendizaje se sigue una aproximación similar, la tasa de aprendizaje durante la primera fase está inicializada a un valor η_0 decreciendo conforme a una función gaussiana y, una vez pasado un número de iteraciones, se fija a un valor η_f.

$$\sigma(t) = \begin{cases} \eta_0 e^{-\frac{t}{\tau}} & si \;\; t < z \\ \eta_f & si \;\; t \geq z \end{cases}$$

Así pues, este algoritmo acerca los pesos del vecindario de la BMU hacia la nueva muestra introducida para parecerse más a la misma. Esto lo hace teniendo en cuenta un vecindario alrededor de la BMU que decrece exponencialmente conforme pasa un número de iteraciones hasta quedarse fijo y una tasa de aprendizaje que también decrece exponencialmente hasta permanecer constante.

Esto permite una primera fase de entrenamiento, con cambios más bruscos en la que se adaptan los valores completamente aleatorios para encontrar agrupamientos razonables. Conforme avanza dicha fase esos valores van decreciendo, hasta que quedan fijados permitiendo a la red neuronal refinar los agrupamientos obtenidos hasta ese momento.

2.1.2. Usos del mapa autoorganizado.

El modelo del mapa autoorganizado puede ser utilizado para diversas tareas, de entre las que destacan:

- **Clustering** - es decir, generar agrupaciones del conjunto de datos de entrada. Por regla general, cada neurona de la capa de Kohonen representaría una posible agrupación de los datos.
- **Visualización de datos de alta dimensionalidad.** Tras finalizar el proceso de entrenamiento, podemos utilizar diferentes técnicas para obtener una representación visual de las características topológicas de la muestras. Las matrices-U, las matrices-P o los planos de componentes son algunos de los modelos utilizados para visualizar el mapa autoorganizado.
- **Clasificación.** Una vez terminado el proceso de entrenamiento, puede asignarse etiquetas a cada uno de los nodos y resolver problemas de clasificación dependiendo de qué BMU se active.

2.1.3. Mapa autoorganizado batch.

El proceso de entrenamiento previamente mencionado se corresponde al del mapa autoorganizado tradicional u *online*. En ese proceso, durante una iteración, se evalúa un subconjunto de los datos como parte de un proceso secuencial de encontrar la BMU y actualizar los pesos correspondientes. Posteriormente, basándose en las propiedades matemáticas del mapa autoorganizado *online*, se derivó una formulación para realizar el proceso de actualización de pesos en una sola iteración para un bloque de muestras. Esta versión del algoritmo, es denominada mapa autoorganizado *batch*.

En esta versión, la regla para la actualización de pesos implica que durante cada iteración, los pesos de las neuronas sean actualizados con la media de las muestras que lo activan teniendo en cuenta los parámetros de control como el vecidario o la tasa de aprendizaje. T_0 representa el inicio de una época y T_f el final de la misma. En cada instante T_k de una época se evalúa una muestra $X(T_k)$del conjunto de datos. La nueva fórmula para la actualización de los pesos es la siguiente:

$$W_{i,j} = \frac{\sum_{k=0}^{f} \delta_f(c,[i,j]) \cdot X(T_k)}{\sum_{k=0}^{f} \delta_f(c,[i,j])}$$

donde c es la unidad de activación (BMU) para la muestra $X(T_i)$ y permitiéndose obviar el parámetro $\eta(t)$ que controlaba la tasa de aprendizaje.

El uso del modelo *batch* frente al modelo tradicional conlleva un intercambio de ventajas e incovenientes [9] que podemos observar en la tabla 2.1.

Ventajas	***Inconvenientes***
Mayores oportunidades de paralelización.	Peor organización topográfica y visualización
Converge más rápido que el tradicional.	
El parámetro η es opcional.	Puede salir clases muy desbalanceadas.
Resultados deterministas, excepto la inicialización si se ha realizado aleatoriamente.	Alta dependencia de la inicialización

Cuadro 2.1: Ventajas e incovenientes de la versión batch

2.1.4. Medidas de calidad.

Para medir la calidad de un mapa autoorganizado una vez entrenado podemos utilizar dos medidas:

El **error medio de cuantificación** nos permite medir la precisión del mapa creado. Se calcula tomando la media de las distancias euclídeas entre cada una de las muestras y su correspondiente BMU.

$$\epsilon_q = \frac{1}{N}\sum_{i=1}^{N} ||x_i - codebook[BMU(x)]||$$

El **error topográfico** mide la capacidad que ha tenido el modelo de conservar las propiedades topográficas del conjunto de muestras de entrenamiento. Podemos medir dicho error como:

$$u(x_k) = \begin{cases} 1 & si\ su\ BMU\ y\ la\ segunda\ BMU\ son\ adyacentes. \\ 0 & en\ caso\ contrario. \end{cases}$$

$$\epsilon_t = \frac{1}{N}\sum_{i=1}^{N} u(x_k)$$

2.2. Árboles de decisión.

Un árbol de decisión [3] es un modelo de aprendizaje automático supervisado utilizado para resolver problemas de clasificación y extensible para resolver problemas de regresión. Un árbol de decisión, una vez entrenado, consiste en una estructura jeráquica de reglas que nos indica a qué categoría pertenece una muestra del conjunto de datos de entrada. Dicho árbol está formado por dos tipos de nodos:

- Los **nodos de decisión**. En dichos nodos existe una pregunta sobre un atributo y valor (o varios) y dependiendo de la respuesta se toma el camino asociado a ésta que parte de este nodo de decisión y nos lleva a otro nodo del árbol.

- Los **nodos terminales** o nodos respuesta nos indican la clase, o el valor en caso de árboles de regresión, a la que ha de pertenecer si al evaluar una muestra dicho nodo ha sido alcanzado. Estos nodos se corresponden con las hojas del árbol formado.

2.2.1. Proceso de entrenamiento.

Obtener un árbol de decisión óptimo es un problema **NP-completo**, es decir, no se conoce manera de resolver este problema con una complejidad de tiempo polinómico por lo que es habitual que los algoritmos que realizan el entrenamiento de este modelo sigan estrategias voraces *(greedy)*. Los algoritmos más conocidos y utilizados para realizar esta tarea(ID3, CART, C4.5, C5.0) siguen un esquema de entrenamiento similar.

La idea que sigue el proceso de entrenamiento es realizar una serie de particiones binarias sobre el conjunto de datos inicial calculando todos los posibles puntos de corte de la partición y evaluando el mejor punto de corte existente. Este proceso es repetido hasta terminar de evaluar el árbol o que alguna de las condiciones de finalización temprana se cumpla, si es que la hubiere.

Todas las posibles divisiones para una partición se basan en repartir todas las muestras basándose en un atributo y un valor asociado a un **punto de corte**. Si, para una muestra, su valor para el atributo en cuestión es inferior al valor del punto de corte, es asignada a una de las subdivisiones y, en caso contrario, es asignada a la otra subdivisión. Habitualmente, los puntos de corte son la media de entre un valor y su sucesor, considerando el conjunto de los valores del atributo ordenado y único, es decir, no se puede

repetir un valor.

Cada una de estas posibles divisiones es evaluada conforme a diferentes criterios dependiendo del algoritmo, habitualmente basados en la impureza de cada una de las dos subdivisiones obtenidas al realizar el corte.

La **ganancia de información** es una de las posibles medidas para determinar el mejor punto de corte de todos los posibles y se basa en la siguiente fórmula:

$$GI(D,s) = Impureza(D) - \frac{|D_{izq}|}{|D|} \cdot Impureza(D_{izq}) - \frac{|D_{der}|}{|D|} \cdot Impureza(D_{der})$$

donde D es el conjunto de datos de la división actual, s es el punto de corte y D_{izq} y D_{der} son las correspondientes subdivisiones obtenidas a partir del punto de corte.

Una medida de **impureza** [10] es una función que, dada un conjunto de datos, mide la cantidad de clases distintas que hay en ese conjunto. Dicha medida valdrá 0 si todos los elementos pertenecen a la misma clase y 1 si cada elemento es de una clase distintas. En la tabla 2.2 destacamos algunas de estas medidas.

Impureza	**Tarea**	**Fórmula**
Entropía	Clasificación	$\sum_{i=1}^{C} -f_i \cdot log(f_i)$
Gini	Clasificación	$\sum_{i=1}^{C} f_i(1-f_i)$
Varianza	Regresión	$\frac{1}{N}\sum_{i=1}^{\|} D\|(y_i - \mu)^2$

Cuadro 2.2: Algunas medidas de entropía.

donde f_i es la probabilidad de pertenecer a la clase i en una división, C es el total de categorías únicas, y_i es el valor del atributo a predecir de una instancia y $\mu = \frac{1}{N}\sum_{i=1}^{N} y_i$ es la media de todas esos valores en una división.

2.2.2. Poda de árboles y criterios de terminación temprana.

Una de las principales cuestiones a la hora de generar un arbol de decisión es conocer cuál ha de ser el tamaño apropiado del mismo para que sea capaz de predecir de la mejor manera posible las muestras con las que se ha entrenado. Un árbol muy pequeño corre el riesgo de haber generalizado más de la cuenta la información procedente de las muestras, mientras que un

árbol muy grande puede estar demasiado especializado, dejarse influir por ruido presente en las muestras y, como consecuencia, producir un sobreajuste de manera que cuando recibe nuevas muestras falla en las predicciones debido a ese gran nivel de especialización sobre el conjunto de entrenamiento.

Para evitar este tipo de problemas, es común recurrir a técncias de podado de árboles. Podemos distinguir dos tipos de técnicas de poda:

- Técnicas de poda realizadas antes de que se termine de generar el árbol *(pre-pruning)*.
- Técnicas de poda realizadas tras la construcción del árbol *(post-pruning)*.

Las técnicas de poda *pre-pruning* ayudan también a que la construcción del árbol finalice antes y, de entre ellas, destacamos:

- Establecer un mínimo de elementos por nodo/partición, de manera que cuando se alcanza dicho umbral esa partición no sigue siendo evaluada.
- Establecer una profundidad máxima del árbol.
- Establecer algún criterio de ganancia de información mínima.

En el momento en que una de estas condiciones se cumple, dicho nodo se convierte en un nodo terminal. En el caso de los problemas de clasificación, es común realizar el voto mayoritario, donde se etiqueta una muestra con la clase más representativa del nodo, es decir la que tiene más instancias en el mismo. Por otro lado, en problemas de regresión es habitual etiquetar el nodo con la media de los valores a predecir (μ) por el mismo.

De entre las técnicas de poda *post-pruning* destacan dos:

La **poda de error reducido**. Esta poda utiliza una técnica simple y rápida de computar en la que, empezando por cada una de las hojas del árbol, se va sustituyendo cada nodo por la clase más popular. Si la predicción no ha empeorado, se continúa en los siguientes niveles de profundidad del árbol asociado al nodo en cuestión y, en el momento en la que dicha predicción empeora, el procedimiento termina.

La **poda de coste-complejidad**. En la poda de coste-complejidad se genera una serie de árboles $T_0, T_1, T_i, ..., T_r$ donde T_0 es el árbol inicial y T_r es sólo la raíz. En cada iteración (i) del proceso, se elimina un subárbol

del árbol anterior $(i-1)$ reemplazándolo con un nodo terminal conforme al siguiente criterio:

$error(T, S)$ es el error del árbol T sobre el conjunto de datos S. $poda(T, t)$ es el árbol obtenido de podar el subárbol t del árbol T.

En cada iteración, se elimina el subárbol que minimiza:

$$\frac{error(poda(T,t),S) - error(T,S)}{|hojas(T)| - |hojas(poda(T,t))|}$$

Una vez generados todos los árboles T_0 a T_r se selecciona aquel que proporciona una mayor precisión.

Generalmente, las técnicas de poda *post-pruning* suelen dar mejores resultados pero son más costosas computacionalmente.

2.2.3. Calidad del modelo.

La principal medida de la calidad del modelo generado aparte de su tiempo de ejecución es la capacidad que tiene de predecir la clase correcta para una muestra. A dicha medida se le denomina **precisión**.

$$Precision = \frac{1}{N}\sum_{i=1} N[1|f(x_i) = y_i]$$

donde f es la función que nos devuelve la clasificación proporcionada por el árbol de decisión, x_i es cada una de las muestras a evaluar e y_i es su correcta clasificación.

2.2.4. Ventajas e incovenientes.

Ventajas.

- Proporcionan un modelo de caja blanca fácil de interpretar y comprender.
- Puede ser combinado con otros modelos distintos o en un conjunto de árboles, por ejemplo, *random forests*.
- No requieren de un gran preprocesamiento de las muestras antes de ser entrenados.

- A diferencia de otros modelos (como la regresión lineal), pueden resolver problemas de clasificación o regresión no lineales.

Incovenientes.

- Pequeños cambios en el conjunto de datos utilizado para el entrenamiento pueden alterar considerablemente los resultados obtenidos.
- Pueden proporcionar peores resultados que otros modelos no lineales (SVM, redes neuronales, etc) con los mismos datos pero, juntarlos en un ensemble, como los *random forest* [11], puede ayudarnos a solventar el problema, eso sí, eliminando la facilidad para interpretar el modelo que proporciona utilizar un único árbol.
- Encontrar el árbol de decisión óptimo global es un problema NP-completo.

Capítulo 3

Implementación.

3.1. Breve introducción a CUDA.

Como comentábamos al principio, CUDA *(Computer Unified Device Arquitecture)* [1] es una tecnología propietaria desarrollada por *NVIDIA* y lanzada en junio de 2007, que nos proporciona de un lenguaje de programación general destinado a ser ejecutado en las tarjetas gráficas de la compañia. Para los propósitos de este trabajo y, habitualmente, a la hora de trabajar con CUDA denominaremos como ***host*** a la CPU que se comunica con la tarjeta gráfica y como **dispostivo** a la GPU o tarjeta gráfica utilizada.

La intercomunicación entre *host* y dispostivo sigue un modelo maestro-esclavo. El *host* actúa como maestro y es el encargado de indicar al dispostivo el código que ha de ejecutar y de mandarlo a la cola del dispositivo. Además, el *host* tiene la posibilidad de trabajar de forma asíncrona con la GPU mientras la cola de trabajos del dispositivo no esté llena.

Es de vital importancia a la hora de trabajar con la GPU de tener en cuenta que:

- a) La GPU tiene muchos más núcleos *(cores)* que una CPU, lo que nos permite realizar mucha más operaciones en el mismo instante. Sin embargo, esto viene a expensas de un menor número de operaciones por segundo de cada núcleo, ya que para distrutar de la cantidad masiva de núcleos que tiene una GPU es necesario que ésta opere a una frecuencia más baja.
- b) La GPU tiene su propia estructura de memoria, que ha de usar para poder realizar operaciones. Dentro de la jerarquía de memoria

encontramos memoria RAM similar a la que utiliza la CPU a través de la placa base, así como varios niveles de caché. Además, hemos de tener en cuenta que a la hora de ejecutar algo en la GPU vamos a tener un gasto extra de tiempo por el traspaso de información de CPU a GPU y viceversa. Minimizar la información que ha de traspasarse en ambos sentidos así como intentar que toda la información necesaria sea transferida a la vez para sacar máximo potencial del PCI Express y exprimir al máximo posible el uso eficiente de la memoria caché, que en CUDA es habitualmente realizado mediante el manejo de la "memoria compartida" es fundamental para obtener mejores resultados, especialmente, aquellos en los que el cuello de botella es la transferencia de datos.

- c) Como la GPU tiene su propia memoria dedicada de un tamaño limitado hemos de hacer hincapié en no utilizar soluciones que generan demasiada complejidad espacial, ya que limitan la escalabilidad de los algoritmos.

3.1.1. Estructura de hebras, bloques y mallas.

El ***kernel*** es un fragmento de código especial destino a ser ejecutado en el dispotivo en el que se indica lo que ha de hacer una hebra.

Las **hebras** son la unidad mínima en la arquitectura CUDA. Cada hebra es ejecutada por un núcleo CUDA. Cada hebra es consciente en tiempo de ejecución de su identificador dentro del bloque así como del identificador del bloque en el que se encuentra y el tamño del mismo, permitiéndonos así repartir el trabajo en función de dichos valores. El **bloque** se corresponde a un conjunto de hebras que ejecuta el mismo *kernel* y que pueden cooperar entre sí y, al conjunto de esos bloques, se le denomina **malla**. Tanto las hebras dentro de un bloque como los bloques dentro de una malla puede tener estructuras unidimensionales, bidimensionales y tridimensionales. Las dimensiones de estas estructuras será indicada por el *host* a la hora de ejecutar el *kernel*.

CUDA exige que un mínimo de 32 hebras, denominado *warp*, ejecuten instrucciones a la vez, aunque se hagan cálculos innecesarios así como que todas las hebras de un bloque sean ejecutadas por el mismo *Streaming MultiProcessor*, de ahora en adelante SM, que es uno de los procesadores en el dispositivo y dispone de un número específico de núcleos CUDA, sus propios registros y su propia caché entre otros.

Al lanzar un *kernel* hemos de utilizar al menos un bloque de N hebras.

Además, en los casos unidimensionales el número de hebras por bloque está limitado a un máximo que depende de la tarjeta gráfica en cuestión.

3.1.2. La memoria compartida.

Dentro de la tarjeta gráfica, nos encontramos con distintos niveles de memoria. Una vez los datos necesarios han sido traspasados del *host* al dispositivo a través del bus PCI Express, esos datos son almacenados en una memoria DRAM de propósito general del dispositivo. Cuando un *kernel* solicita datos de esta memoria, de manera similar a como ocurre en una CPU, los datos solicitados y los colidantes en memoria son colocados a través de varios niveles de caché, que tiene tamaño más limitado que la memoria DRAM pero con un acceso de lectura y escritura mucho más rápido.

La **memoria compartida** es una abstracción para una región especial de la caché asociada a un bloque que es explícitamente usada por el programador en el *kernel*, agilizando así considerablemente las transferencias de memoria en el dispositivo. En el cuadro 3.1, podemos ver un resumen de los tipos de memoria existentes, dónde se pueden usar y dónde se encuentran dichos datos en el dispositivo.

Memoria	**Localización**	**Acceso (E = Escribir) (L = Leer)**	**Existente hasta fin de**
Registro	Caché	Kernel (E/L)	Hebra
Local	DRAM (Caché tras uso)	Kernel (E/L)	Hebra
Compartida	Caché	Kernel (E/L)	Bloque
Global	DRAM (Caché tras uso)	Host (E/L) Kernel (E/L)	Aplicación o uso de free
Constante	DRAM (Caché tras uso)	Host (E/L) Kernel (L)	Aplicación o uso de free

Cuadro 3.1: Resumen de los tipos de memoria en CUDA.

3.1.3. Python: Numba y CuPy.

Para desarrollar el código asociado a este proyecto, hemos optado por utilizar **Python** en vez de los tradicionales C o C++. El uso de *Python* nos permite un desarrollo de los algoritmos más rápido así como el acceso a abstracciones de más alto nivel mediante el uso de la librerías ***Numba*** y ***Cupy***, así como una mayor facilidad para la distribución del código, si

se desea, mediante el uso de *PyPI(Python Package Index)*, el repositorio de paquetes para Python, .

Numba [12] es un paquete para Python cuyo objetivo es la aceleración compilado fragmentos de código utilizando el compilador LLVM y dando la oportunidad de paralelizar código tanto para la CPU como para la GPU. En concreto, para las GPUs CUDA, proporciona al usuario un subconjunto de las características de CUDA con un nivel de abstracción mayor. Con eso no sólo conseguimos poder trabajar con CUDA desde Python sino, también evitar, si lo deseamos, manejar los traspasos de memoria entre host y dispositivo o la necesidad de indicar todos los tipos a la hora de inicializar un *kernel* entre otras ventajas.

```
from numba import cuda
import numpy as np
# Definimos el kernel
@cuda.jit
def aumentar_en_1(un_array):
  # Cogemos el índice de la hebra
    pos = cuda.grid(1)

    # Si el índice está en el rango del array
    # incrementamos su valor
    if pos < un_array.size:
        un_array[pos] += 1

if __name__ == '__main__':
  # Declaramos un array de 10000 ceros
  ejemplo = np.zeros(10000)
  # Calculamos el número de bloques necesario
  bloques = ejemplo.size // 128 + 1
  # Lanzamos el kernel con bloques de 128 hebras
  aumentar_en_1[bloques, 128](ejemplo)
```

Código Fuente 3.1: Kernel para incrementar en 1 los elementos de un array.

CuPy [13] es otro paquete de Python que, por un lado y de manera similar a Numba, nos permite generar kernels para CUDA en este caso de manera similar a los de C/C++ así como facilidades para generar kernels en los que se implementa reducciones u operaciones elemento a elemento en un array. Por otro lado, proporciona una API similar a la de NumPy pero las operaciones están implementadas utilizando CUDA. Además, CuPy está implementado de manera que permite utilizar directamente sus estructuras de datos sobre kernels de Numba, lo que nos permite combinar elementos de ambos paquetes según nos interese.

3.1.4. Spark.

Apache Spark es un *framework* de código abierto y propósito general para sistemas distribuidos de computación en clúster que proporciona una API utilizable desde los lenguajes de programación en Scala, Java, Python y R. El *framework* fundamenta su arquitectura en el *RDD (Resilient Distributed DataSet)*, que es una estructura de datos de sólo lectura distribuida en un clúster de máquinas, mantenida durante toda la computación y con tolerancia a fallos. Además, proporciona otras herramientas de alto nivel como ML/MLib, una librería con algoritmos de *machine learning*.

Utilizando la API de Python, podemos combinar el uso de *Spark* y *Numba CUDA* para afrontar problemas de grandes dimensiones, ya que el *RDD* nos permite trabajar con subconjuntos de esos datos posibilando incluso llevar las lementaciones realizadas a un clúster con múltiples sistemas con dispositivos GPU *CUDA* cont todas las dependencias necesarias instaladas.

La distribución de trabajo en Spark se realizará utilizando la transformación *mapPartitions* del *RDD* de *Spark*, que generá un nuevo RDD a partir de los resultados obtenidos al aplicar la función pasada a *mapPartitions* como parámetro a cada una de las funciones.

3.2. Proceso de implementación.

Para realizar la implementación de cada algoritmo hemos realizado un proceso cíclico divido en 3 fases:

- **Análisis** - En la primera iteración, analizar los trabajos relacionados. En las posteriores, analizar los resultados obtenidos del profiler, determinar los cuellos de botella y buscar posibles alternativas para solucionar el problema.
- **Implementación** - Realizar la implementación en CUDA de los cambios o elementos nuevos obtenidos del proceso de análisis.
- **Profiling** - Utilizar el profiler de NVIDIA, *nvprof*, sobre un ejemplo razonable para evaluar el rendimiento del algoritmo.

En este capítulo, explicaremos las soluciones finales a las que hemos llegado y destacaremos algunas de las decisiones tomadas y el razonamiento para haberlas seleccionado. Para facilitar la comprensión y tener en cuenta las dependencias de datos de los algoritmos durante el proceso de desarrollo

hemos utilizado diagramas de flujo adaptados que nos permitan tener una visión general de las dependencias existentes entre los procesos y datos de una manera esquemática para cada uno de los modelos.

3.3. Desarrollo del mapa autoorganizado de Kohonen.

Para implementar los mapas autoorganizados de Kohonen, primero consideramos la versión tradicional *online* y, posteriormente, tras ver las limitaciones de la primera, evaluamos la versión computada en *batchs*. Ambas implemntaciones ha sido realizadas tanto para CPU secuencial (un núcleo) como para CUDA. El código desarrollado en la versión para CPU secuencial ha sido realizado en *Python* usando *NumPy* conforme a lo explicado al presentar el modelo. En este capítulo, nos centramos en la implementación realizada en *CUDA*.

3.3.1. Limitaciones del mapa autoorganizado online.

En la figura 3.1 podemos observar un esquema de cómo podríamos resolver el mapa autoorganizado online utilizando CUDA.

Mientras que este fue el punto de partida para la realización de este trabajo tuvimos que descartar esta versión del algoritmo. Mientras que nosotros buscamos resolver problemas con un gran número de muestras utilizando *CUDA* y *Spark*, dicho algoritmo requiere que una única muestra sea evaluada por iteración, ya que cada muestra, al ser evaluada, modifica los pesos para la siguiente iteración en la que otra muestra realizará el mismo proceso. Por ello, esta opción ha sido descartada aunque si nos encontrásemos ante un problema con un número de neuronas superior al número de muestras de entrada, factor que no parece natural en un algoritmo cuyo principal uso es el *clustering*, el algoritmo es susceptible a ser paralelizado en base al número de neuronas del mapa.

Determinada esa limitación fundamental, vamos a centrarnos en la versión del mapa autoorganizado que hemos implementado, el mapa autoorganizado *batch*.

3.3.2. Desarrollo del mapa autoorganizado batch.

Para el desarrollo de esta versión del algoritmo de una manera eficiente dos factores han sido clave: el **uso de la memoria compartida** y **la reducción**. Podemos descomponer la implementación del algoritmo en resolver los

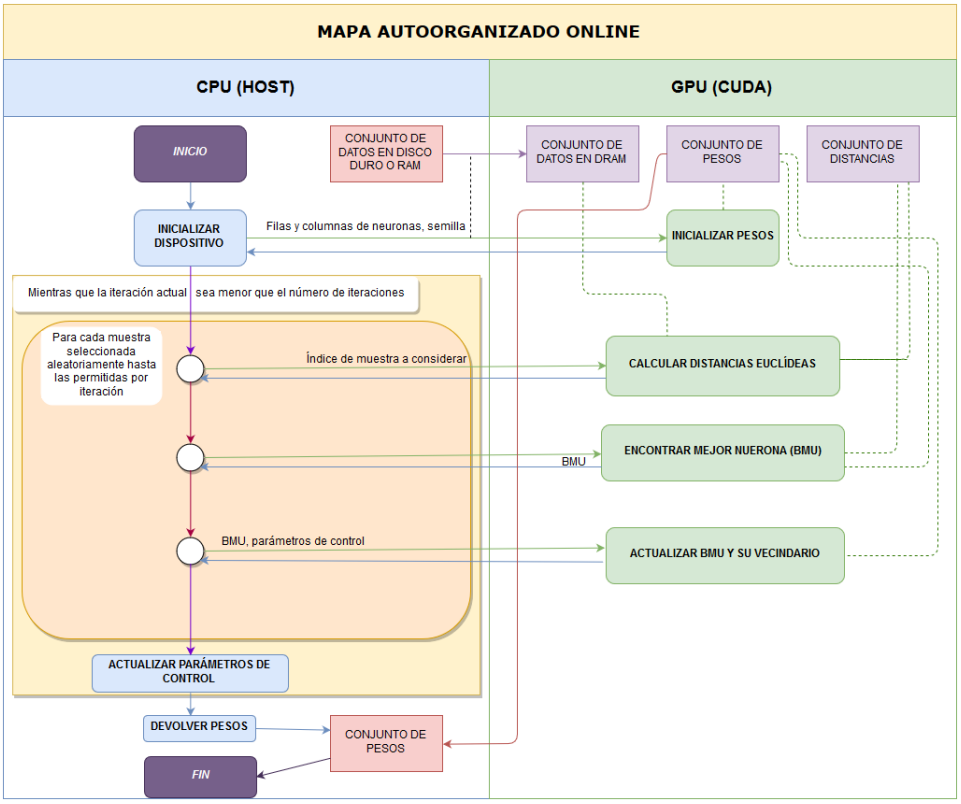


Figura 3.1: Diagrama de flujo para el mapa autoorganizado online.

siguientes subproblemas:

- 1. Inicialización aleatoria de la matriz de pesos.
- 2. Cálculo de las distancias euclídeas entre todas las muestras y los pesos.
- 3. Encontrar la BMU para cada muestra.
- 4. Actualizar la matriz de pesos.
- 5. Actualizar los parámetros de control.

3.3.2.1. Inicialización aleatoria de la matriz de pesos.

Este proceso es realizado por el nodo de Spark que controlaría la ejecución del clúster una única vez al inicio del algoritmo, pero utilizando la GPU. *Numba CUDA* nos proporciona herramientas para la generación de

valores flotantes en el rango comprendido entre 0 y 1 basadas en el método de Box-Muller. Hemos utilizado esta herramienta para la generación de nuestra matriz de vectores de pesos inicial.

```
@cuda.jit
def cuda_init_weights(rng_states, weights):
    idx = cuda.grid(1)
    if idx < weights.size:
        weights[idx] = xoroshiro128p_uniform_float32(rng_states, idx)

d_weights = cuda.device_array(rows * cols * d, np.float32)
rng_states = create_xoroshiro128p_states(rows * cols * d, seed=seed)
cuda_init_weights[rows * cols * d // tpb + 1, tpb](rng_states, d_weights)
```

Código Fuente 3.2: Incialización aleatoria de la matriz de pesos.

Una vez estos pesos han sido generados se realiza un proceso iterativo con los subproblemas 2 al 5 hasta alcanzar el número de iteraciones máximo determinado por el usuario. Cada nodo ejecutor de Spark se encarga de calcular los resultados sobre un subconjunto de muestras, combinándolos al final de la iteración el nodo que lleva el control de la ejecución. Este proceso es realizado utilizando la función *mapPartitions* de *Spark* que nos permite realizar la misma operación sobre las diferentes particiones del *RDD* que contiene los datos y obtener otro *RDD* reuniendo los resultados obtenidos de la operación.

3.3.2.2. Cálculo de las distancias euclídeas entre todas las muestras y los pesos.

El cálculo de la distancia euclídea es una operación masiva uno a uno realizada con todas las muestras de entrada con cada uno de los vectores que representa los pesos de las neuronas del mapa. Para optimizar este operación en *CUDA* hemos realizados dos optimizaciones. Por un lado, puesto que nuestro objetivo final es encontrar la posición de la combinación muestra-neurona con la distancia mínima podemos eliminar el cálculo de la raíz cuadrada de la fórmula de la distancia euclídea pues es una operación costosa y no altera la relación de orden generada. Por otro lado, utilizamos la memoria compartida para asegurarnos de que algunos elementos que son ampliamente utilizados en el bloque permanezca en esa caché de acceso más rápido que la memoria global. En concreto, en este caso, lanzamos una malla bidimenisonal de bloques con tantas filas como número de neuronas y columnas como el número de muestras dividido por el tamaño de un bloque más una. Dada esta distribución, en el mismo bloque estamos calculando la distancia entre la neurona con el asociado a la fila en la malla y el subcojunto de muestras asociada a la columna de la malla. Puesto que los pesos de la

neurona son utilizados en todo su bloque, éstos son cargados en la memoria compartida del bloque.

```
@cuda.jit
def euclidean_distance(samples, weights, distances, nsamples, d):
    # 1. Tomamos los índices que nos correspoden
    neuron_idx = cuda.blockIdx.x
    samples_idx = cuda.blockIdx.y * cuda.blockDim.x + cuda.threadIdx.x


    # 2. Ponemos los pesos de la neurona en memoria compartida
    shared_weights = cuda.shared.array(shape=0, dtype=numba.float32)
    for i in range(d // cuda.blockDim.x + 1):
        i_stride = i * cuda.blockDim.x
        my_pos = i_stride + cuda.threadIdx.x
        if my_pos < d:
            shared_weights[my_pos] = weights[neuron_idx * d + my_pos]

    cuda.syncthreads()

    # 3. Procedemos a realizar el cálculo de la distancia si procede
    if samples_idx < nsamples:
        distance = 0.0
        for i in range(d):
            i_distance = samples[samples_idx * d + i] - shared_weights[i]
            distance += i_distance * i_distance

        distances[samples_idx, neuron_idx] = distance
```

Código Fuente 3.3: Cálculo de la distancia euclídea.

3.3.2.3. Encontrar la BMU para cada muestra.

Todas las distancias del subproblema anterior fueron guardadas en una matriz bidimensional donde la fila representa a la muestra y la columna a la neurona. Por tanto, hemos de encontrar el valor en mínimo en cada fila.

Para encontrar el mínimo en un *array* utilizamos un algoritmo frecuentemente utilizando en la GPU: **la reducción**. Si los elementos de la reducción, caben en un bloque, los mismos son puestos en la memoria compartida del bloque y se simula un recorrido sobre un árbol binario balanceado hacia arriba, tomando los elementos cargadas en memoria compartida como las hojas y alcanzado el resultado final en la raíz. Para que el algoritmo funcione, la operación en cuestión ha de cumplir la propiedad asociativa. Mientras que el uso más habitual de este algoritmo es para realizar la sumatoria nosotros la utilizamos para encontrar el mínimo manteniendo también control del índice que le corresponde. Si todos los datos no caben en un bloque, se lanzan tantos bloques como sean necesarios y este proceso es repetido hasta

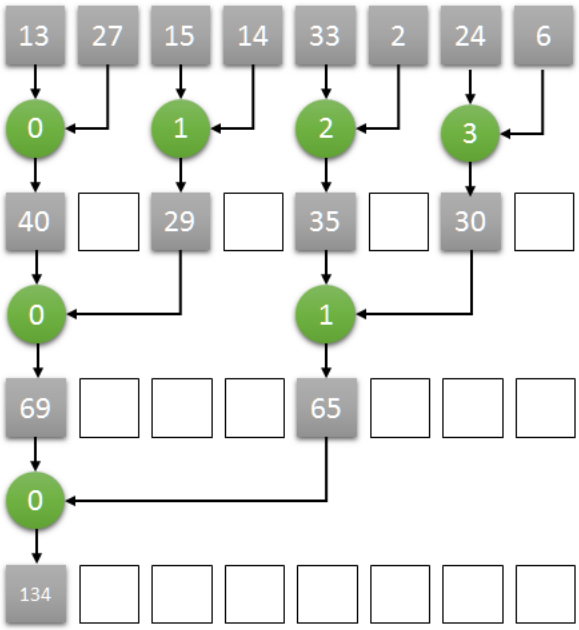


Figura 3.2: Una reducción paralela de una sumatoria en CUDA.

que queda un único resultado.

Puesto que nosotros queremos realizar el proceso para múltiples filas con las mismas dimensiones. Si necesitamos P bloques para resolver la reducción de una fila lanzamos los $N \cdot P$ correspondientes a la vez para obtener la máxima ganancia. Podemos consultar con más detalle cómo realizar una implementación de una reducción de alto rendimiento en *CUDA* en la referencia bibliográfica [14].

3.3.2.4. Actualizar la matriz de pesos.

La fórmula para actualizar cada vector de pesos del mapa depende de dos sumatorias, una en el numerador y otra en el denominador.

$$W_{i,j} = \frac{\sum_{k=0}^{f} \delta_f(c,[i,j]) \cdot X(T_k)}{\sum_{k=0}^{f} \delta_f(c,[i,j])}$$

Para utilizar los múltiples nodos ejecutores de *Spark* y sacar el máximo rendimiento, cada nodo realiza la sumatoria de sus muestras guardando el numerador y denominador parciales que les corresponden a sus particiones para, a continuación, ser combinados en el nodo de *Spark* que dirige la eje-

cución del algoritmo.

El cálculo parcial de los numeradores y denominadores parciales se realiza utilizando **operaciones atómicas**, en concreto, la suma atómica. Las operaciones atómicas son operaciones que sufren de una mayor latencia que su correpondiente operación normal pero evitan condiciones de carrera si múltiples hebras intentan modificar la misma posición de memoria pues realizan la lectura y la actualización del dato en una única operación. La necesidad de utilizar este tipo de operaciones radica en la posibilidad que existe de que la BMU de varias muestras sean la misma neurona y, por tanto, varias hebras intenten modificar la sumatoria de numerador o denominador de la misma neurona al mismo tiempo. Además de esta alternativa, que es por la que hemos optado para realizar el desarrollo de esta parte del algoritmo, podríamos haber generado una estructura auxiliar que para cada neurona contenga $N \cdot d$ entradas para la sumatoria del numerador y otras N para la sumatoria del denominador y posteriormente realizar una reducción con la operación de suma para obtener los resultados deseados. Sin embargo, esta alternativa, como podemos observar, sufriría de una gran complejidad espacial a la hora de afrontar problemas de *Big Data*, limitando la escalabilidad de la solución propuesta a la hora de afrontar problemas de mayor número de muestras o mapas de neuronas más grandes.

Para lanzar nuestro kernel aprovechando la memoria compartida usamos una malla de tantas filas como números de muestras y tantas columnas como números de neuronas haya en el mapa divido entre el tamaño del bloque más una. Puesto que los bloques de cada fila de la malla, comprueban si la muestra de la fila está en el rango de actualización de la neuronas que le corresponde a cada bloque mantenemos los datos de la muestra en memoria compartida para asegurarnos de un acceso rápido a los mismos.

```
@cuda.jit
def prepare_update(bmu, samples, num, den,
    nrows, ncols, d, sigma_squared):
    # 1. Tomamos los índices que correspondan
    sample_idx = cuda.blockIdx.x
    neuron_idx = cuda.blockIdx.y * cuda.blockDim.x + cuda.threadIdx.x

    # 2. Metemos en memoria compartida la muestra que se lee en todo el bloque
    shared_sample = cuda.shared.array(shape=0, dtype=numba.float32)
    for i in range(d // cuda.blockDim.x + 1):
        i_stride = i * cuda.blockDim.x
        my_pos = i_stride + cuda.threadIdx.x
        if my_pos < d:
            shared_sample[my_pos] = samples[sample_idx * d + my_pos]
    cuda.syncthreads()

    # 3. Si procede realizar cálculos los hacemos
    if neuron_idx < nrows * ncols:
```

```
        bmu_row = bmu[sample_idx] // ncols
        bmu_col = bmu[sample_idx] % ncols
        neuron_row = neuron_idx // ncols
        neuron_col = neuron_idx % ncols

        dist = (neuron_row - bmu_row) * (neuron_row - bmu_row) + \
               (neuron_col - bmu_col) * (neuron_col - bmu_col)

        if dist <= sigma_squared:
            hck = math.exp(-dist/(2 * sigma_squared))
            # Guardamos sumatoria del denominador
            cuda.atomic.add(den, neuron_row * ncols + neuron_col, hck)
            # Guardamos sumatoria del numerador
            for i in range(d):
                cuda.atomic.add(num, neuron_row*ncols*d + neuron_col*d+i
                                hck * shared_sample[i])
```

Código Fuente 3.4: Cálculo de numeradores y denominadores parciales.

Una vez todos los resultados han sido recopilados, lanzamos un último kernel que juntará los resultados parciales obtenidos y realizará la división, actualizando los pesos si la neurona en cuestión ha sido BMU de alguna muestra o dejando los pesos de la iteración anterior en caso contrario. En este caso lanzamos una malla unidimensional de tantos bloques como número de neuronas haya entre el tamaño de un bloque más uno. Cada hebra se encarga de realizar las sumas parciales de su neurona asociada y realizar la actualización de su vector de pesos en la matriz si procede.

```
@cuda.jit
def finish_update(weights, partials, numParts, nrows, ncols, d):
     idx = cuda.grid(1)
    if idx < nrows * ncols:
        row = idx // ncols
        col = idx % ncols


        # a) Sumamos todos los parciales en el primer array
        numsize = nrows * ncols * d
        densize = nrows * ncols
        fullsize = numsize + densize
        for i in range(numParts - 1):
            # Suma de numeradores
            for k in range(d):
                pos = fullsize * i + row * ncols * d + col * d + k
                partials[row * ncols * d + col * d + k] += partials[pos]
            # Suma de denominadores
            pos = fullsize * i + numsize + row * ncols + col
            partials[numsize + row * ncols + col] += partials[pos]

    cuda.syncthreads()

    if idx < nrows * ncols:
```

```
# b) Si no es 0 el denominador realizamos el cambio
if partials[numsize + row * ncols + col] != 0:
    for k in range(d):
        mypos = row * ncols * d + col * d + k
        denpos = numsize + row * ncols + col
        weights[mypos] =  partials[mypos] / partials[denpos]
```

Código Fuente 3.5: Actualización final de la matriz de pesos.

3.3.2.5. Actualizar los parámetros de control.

Los parámetros de control η y σ son sólo dos parámetros a modificar por iteración y cuyo cálculo se corresponde a una única operación, por lo que nos es susceptible a ser paralelizado en la GPU y será realizado en el nodo de Spark que dirige la ejecución del clúster si todavía no se ha alcanzado el máximo de iteraciones. Si se ha alcanzado el máximo de iteraciones el algoritmo termina y la matriz de vectores de pesos de las neuronas de esa última iteración es la solución obtenida.

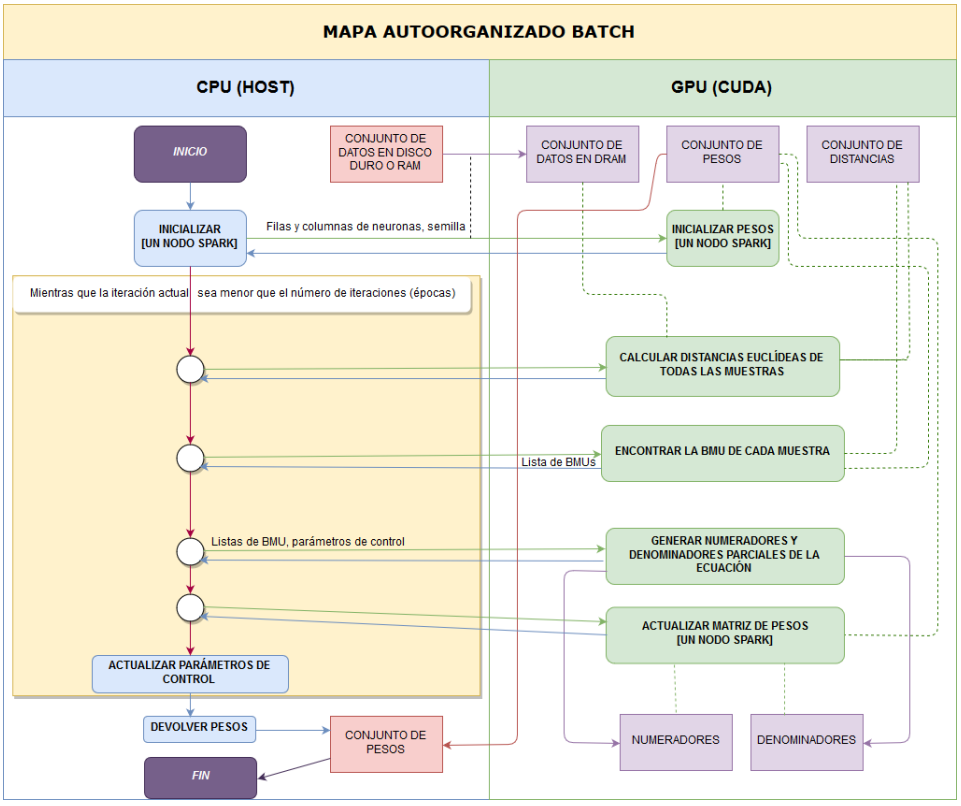


Figura 3.3: Diagrama de flujo para el mapa autoorganizado batch.

3.4. Desarrollo de un modelo de árbol de decisión.

La implementación del modelo de árbol de decisión se basa en CUDT [7], que a su vez se fundamenta en SPRINT [15] y la operación de *scan*.

3.4.1. Lista de atributos.

Una lista de atributos, es una estructura auxiliar, procedente de SPRINT [15], utilizada para representar las clases y los atributos asociados a una muestra. Una lista de atributos tiene una estructura similar a la siguiente tabla:

Valor	Clase	ID Muestra
2,5	0	0
4,7	0	1
0,1	1	2
1,0	1	3

Cuadro 3.2: Una lista de atributos sin ordenar.

Las columnas de la tabla 3.2 son:

- **Valor**, que se corresponde al valor que toma el atributo al que corresponde la tabla en la muestra representada en la fila.
- **Clase**, que se correponde a la etiqueta de salida asoaciada a la muestra de la fila.
- **ID Muestra**, que se correponde al identificador de la muestra. Al principio, se correponde al número de fila empezando por 0.

Una vez esta estructura es generada para cada atributo del problema en cuestión, es ordenada por orden creciente según la columna "Valor".

3.4.2. Esquema general del algoritmo implementado.

Al inicio del algoritmo, tras generar las listas de atributos, se genera un nodo raíz que comprende todas las muestras del conjunto. En ese nodo, hemos de encontrar para qué atributo y qué valor realizamos la partición óptima de los datos. Para ello, se consideran todas las listas de atributos y se toma como posible punto de corte el punto medio entre un valor y el siguiente si no se trata del mismo valor. Asociada a cada una de las particiones, se calcula el criterio de Gini. Una vez realizados todos los cálculos, tomamos

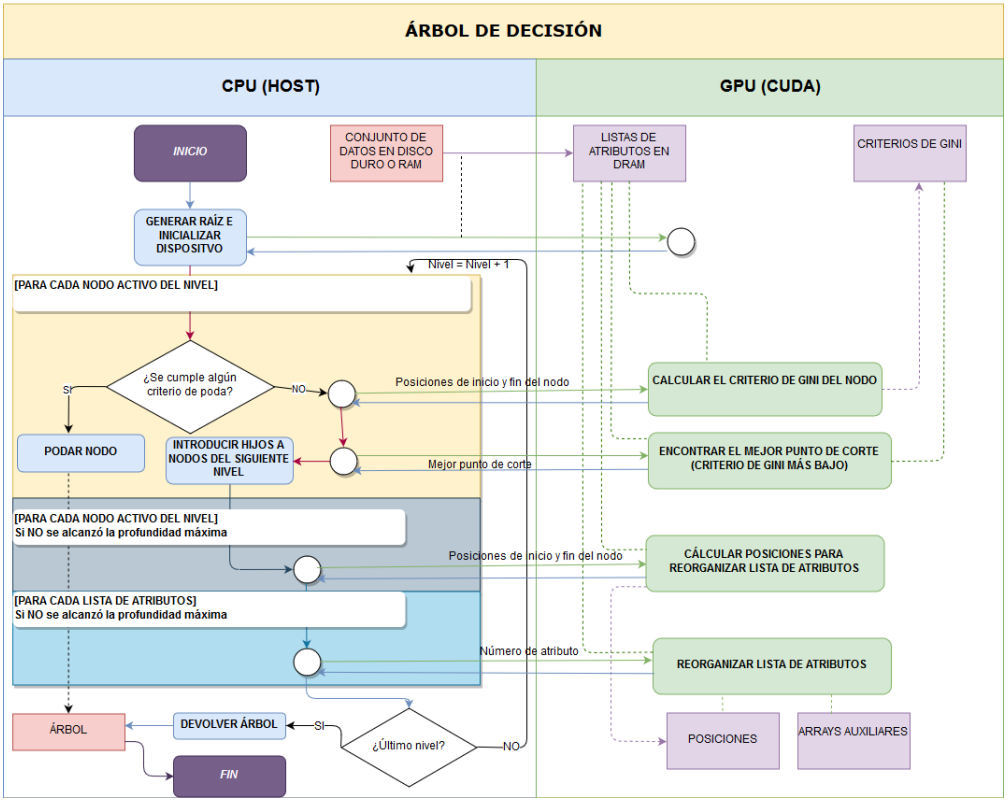


Figura 3.4: Diagrama de flujo de la implementación del árbol de decisión.

como punto de corte aquella que menor criterio de Gini nos de. Utilizando ese punto de corte, generamos dos nuevos nodos para la siguiente iteración, uno que contiene todos los puntos menores o iguales que dicho punto de corte y otro con los mayores. Además, dicho punto es el que utilizamos para generar nuestro nodo de decisión en el árbol entrenado. Este proceso se repite hasta que no quedan nodos por evaluar. Un nodo no ha de ser evaluado si:

- a) Todos los elementos del nodo pertenecen a la misma clase. En ese caso, en vez de un nodo de decisión, generamos un nodo terminal con la clase correspondiente.
- b) Se ha especificado un criterio de profundidad máxima y dicha profundidad ha sido alcanzada. En ese caso, generamos un nodo terminal con la clase más representativa del nodo.
- c) Se ha especificado un límite para el número de elementos mínimo que puede contener y ha sido alcanzado. En ese caso, generamos un nodo terminal de manera similar al caso anterior.

3.4.3. La operación de scan.

Una de las claves del uso de la lista de atributos, es que, para los problemas de **clasificación binaria**, que son los únicos que nuestro modelo es capaz de resolver, si codificamos una clase como 0 (a partir de ahora llamada *clase negativa*) y otra como 1 (*clase positiva*) si realizamos una suma acumulada sobre el subconjunto de filas de un nodo de la columna "Clase" podríamos tener control de cuántos elementos hay en cada clase tanto para todas las particiones. Para realizar la suma acumulada existe un algoritmo ampliamente utilizada en el mundo de la GPU denominado *scan*.

El *scan* [16], suma acumulada o suma prefija, es una operación que utiliza un operador binario, $\oplus$, que cumpla la propiedad asociativa y utilizada sobre un *array* de n elementos. Existen dos formas de realizar el *scan*: inclusivo y exclusivo. El scan inclusivo empieza con el primer elemento del array y va a realizando una suma acumulada. El *scan* exclusivo empieza con el elemento neutro de la operación y realiza una suma acumulada de todos los elementos hasta el penúltimo. En la implementación realizada hemos utilizado una operación de *scan* exclusivo que además devuelve el total de la suma del array cubriendo ambos casos. Para los propósitos de este documento, cada vez que hablemos de *scan* nos estaremos refiriendo al *scan exclusivo*. Por tanto la operación de *scan* sobre un array nos devuelve lo siguiente:

$$scan([a_0, a_1, a_2, ..., a_{n-1}]) = [I, a_0, (a_0 \oplus a_1), (a_0 \oplus a_1 \oplus a_2), ..., (a_0 \oplus a_1 \oplus a_2 \oplus ... \oplus a_{n-1})]$$

La implementación en *CUDA* de la operación se realiza, utilizando una técnica frecuente para estos dispositivos, simular el uso de un árbol binario balanceado utilizando la memoria compartida del bloque. Imaginamos que todos los elementos de un bloque son los nodos terminales de un árbol y los metemos en la memoria compartida del bloque y realizamos dos fases sobre ello. En la primera fase, *up-sweep* (figura 3.5), se realiza el recorrido de los nodos terminales a la raíz realizando las sumas en sus nodos padre y manteniendo las sumas paciales. En la segunda fase *down-sweep* (figura 3.6), se utilizan los resultados parciales obtenidos para completar los resultados en la suma acumulada. Para arrays cuya dimensión no cabe en un bloque dicho array es subdividido en múltiples subconjuntos contiguos del mismo tamaño que sí caben en un bloque (si sobran elementos en el último bloque son inicalizados con el elemento neutro de la operación) y se genera el scan sobre todos los subconjuntos así como se almacena la sumatoria (puesto en que nuestra caso la operación es la suma) de todos los elementos del bloque. Una vez finalizado, dichas sumatorias se aplican a los bloques que le preceden en posición, obtiendo el resultado final deseado.

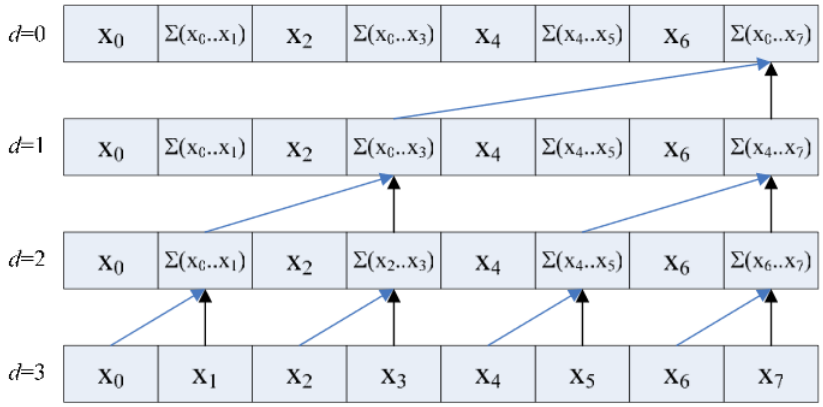


Figura 3.5: Fase up-sweep del scan.

3.4.4. Cálculo del criterio de Gini.

Dado que sólo vamos a calcular el criterio de Gini para problemas de clasificación binaria hemos simplificado el mismo para ahorrarnos algunas operaciones a la hora de realizar el cálculo:

$$CRITERIO(A, v) = \frac{|i : A_i \leq v|}{N} \cdot GINI(|i : A_i \leq v|) + \frac{|i : A_i > v|}{N} \cdot GINI(|i : A_i > v|)$$

$$GINI(D) = 1 - \frac{T_D^2}{N_D^2} - \frac{F_D^2}{N_D^2}$$

Siendo N_D el total de las muestras en el nodo D, T_D el total de muestras pertenecientes a la clase positiva y F_D el total de muestras pertenencientes a la clase negativa. Tenemos que:

$$F_D = N_D - T_D$$

Sustituyendo obtenemos que:

$$GINI(D) = \frac{N_D^2 - T_D^2 - (N_D - T_D)^2}{N_D^2} = \frac{N_D^2 - T_D^2 - (N_D^2 + T_D^2 - 2N_D T_D)}{N_D^2}$$

$$GINI(D) = \frac{-2T_D^2 + 2N_D T_D}{N_D^2} = 2\frac{T_D(N_D - T_d)}{N_D^2}$$

$$CRITERIO = \frac{N_\leq}{N}\frac{2T_\leq(N_\leq - T_\leq)}{N_\leq^2} + \frac{N_>}{N}\frac{2T_>(N_> - T_>)}{N_>^2}$$

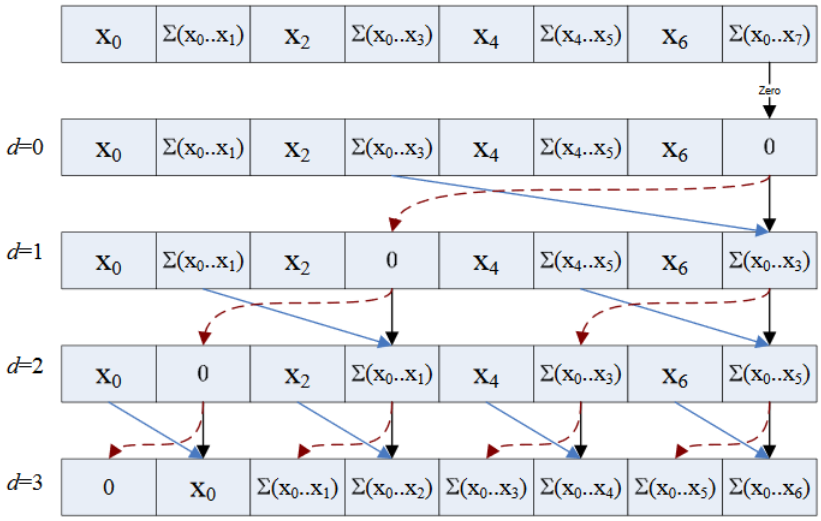


Figura 3.6: Fase down-sweep del scan.

$$CRITERIO = \frac{2}{N}\left(\frac{T_{\leq}(N_{\leq} - T_{\leq})}{N_{\leq}} + \frac{T_{>}(N_{>} - T_{>})}{N_{>}}\right)$$

Puesto que además, no es de nuestro interés el valor específico sino obtener el valor óptimo, podemos ahorrarnos la multiplicación por $\frac{2}{N}$. Así pues, calculamos el criterio de la siguiente manera:

$$CRITERIO' = \left(\frac{T_{\leq}(N_{\leq} - T_{\leq})}{N_{\leq}} + \frac{T_{>}(N_{>} - T_{>})}{N_{>}}\right)$$

El valor de $CRITERIO'$ oscilará entre 0 y $\frac{N}{2}$ y buscaremos siempre obtener el mínimo valor para este criterio. Dicha búsqueda se realizará de manera similar a la realizada para el modelo anterior con una reducción para encontrar el índice mínimo en cada nodo.

3.4.5. Reorganización de la listas de atributos.

Para finalizar la evaluación de los nodos de un nivel podemos volver a aprovechar la operación de *scan* para reorganizar el orden de los elementos de la lista de atributos sin necesidad de ejecutar ningún algoritmo de ordenación. Una vez se ha seleccionado la combinación de mejor lista de atributos para un nodo y su punto de corte, puesto que esta lista ya estaba ordenada, todos los elementos hasta el punto de corte pertenecen al nodo hijo izquierdo y los posteriores al nodo hijo derecho. Además, puesto que tenemos en la lista de atributos el cammpo “ID Muestra”, podemos generar fácilmente

un array de booleanos donde cada elemento indica si la muestra con el ID asociado a su posición pertenece al hijo izquierdo o al hijo derecho. Una vez hecho esto, podemos recorrer cada nodo activo de la lista de atributos y utilizar este array auxiliar y la operación de scan para determinar la nueva posición que va a ocupar el elemento dentro del nodo. Esto se realiza, aplicando un scan sobre las posiciones de este array que le corresponden a cada muestra del nodo que va a ir a la subdivisión izquierda. De esta manera, si la muestra pertenece a la subdivisión izquierda su nueva posición será la suma acumulada de elementos de la subdivisión izquierda menos uno (porque empezamos a indexar el array en 0) y en caso de pertenecer a la subdivisión de la derecha será la posición más la diferencia del total de elementos de la subdivisión izquierda (suma acumulada total) con la suma acumulada de elementos de la subdivisión izquierda menos uno.

3.4.6. Limitaciones y uso de Spark.

Como comprobaremos posteriormente, este modelo, al requerir de la evaluación independiente de múltiples nodos y no presentar técnicas de poda avanzadas no escala bien con la generación de árboles profundos o completos. Es por eso que, para afrontar problemas de mayores dimensiones, en vez de generar un único árbol, vamos a generar tantas particiones del *RDD* de *Spark* como árboles deseemos y montar un modelo de *random forest*, en el que en cada partición tenemos un conjunto aleatorio de muestras tomadas sin reemplazo y consideramos todos los atributos para todos los árboles. La clasificación proporcionada por el modelo viene dada por el clase mayoritariamente votada por el conjunto de árboles.

Capítulo 4

Desarrollo de pruebas y análisis de resultados.

4.1. Entorno de pruebas.

Para el desarrollo de las pruebas, mi ordenador personal ha sido utilizado. Las especificaciones ténicas relevantes del mismo son:

- **Placa Base:** MSI B450M Bazooka
- **Sistema Operativo:** Windows 10 64 bits
- **CPU:** AMD Ryzen 5 2600X
- **RAM:** Kingston HyperX Fury Black DDR4 2400 MHz PC4-19200 8GB CL15
- **GPU**: Zotac GeForce GTX 1060 AMP! Edition

  Núcleos CUDA: 1280

  Frecuencia del procesador: 1556 MHz (1771 MHz Boost)

  Frecuencia de la memoria: 8 GHz

  Memoria: 6 GB DDR5

  Bus de memoria: 192-bit

  Compute Capability: 6.1

4.2. Conjuntos de datos utilizados.

Durante la fase de desarrollo del mapa autoorganizado hemos utilizado el conjunto de datos de las **caras de Olivetti**, creado por *AT&T Laboratories Cambridge* y descargada a través del paquete de Python *scikit-learn* [17]. Dicho conjunto de imágenes consiste en 400 imágenes de 40 sujetos en escala de grises. Cada muestra son los valores de intensidad de cada píxel con un valor normalizado entre 0 y 1. Además, se proporciona una etiqueta que indica a qué sujeto pertenece cada imagen pero para los propósitos de nuestro modelo de aprendizaje no supervisado la misma no será utilizada. Las imágenes están en una versión cuadrada de 64x64 píxeles dándonos un total de 4096 valores de intensidad por muestra.

Durante la fase de desarrollo del árbol de decisión hemos utilizado dos conjuntos de datos de problemas de clasificación binaria: **Spambase** y **MAGIC Gamma Telescope**.

Spambase [18] es un conjunto de 4601 muestras con 57 atributos. El objetivo en este conjunto de muestras es diferenciar correos no deseados (*spam*) de correos deseados en función de los 56 atributos numéricos basados en el contenido del correo electrónico asociado a la muestra.

Magic Gamma Telescope [19] es un conjunto de 19020 muestras con 11 atributos. Los datos de este conjunto fueron obtenidos en un experimento con un telescopio especial para observar rayos gamma de alta energía. El objetivo es diferenciar imágnes tomadas por el telescopio y preprocesadas de estas muestras generadas por rayos gamma (*signal*) de las de rayos cósmicos en la capa superior de la atmósfera (*background*). Los datos de este conjunto fueron generados a partir de simulaciones de Monte Carlo.

Para evaluar el rendimiento de ambos modelos para conjuntos de *Big Data* hemos utilizado **SUSY** [20]. Este conjunto de datos contiene 5 millones de muestras con 18 atributos, que se generó a partir de un experimento de física en el que también se intenta diferencia un proceso que genera partículas supersimétricas (*signal*) de otro proceso que no las genera (*background*). En el caso del mapa autoorganizado la clase de salida es ignorada. De manera similar al anterior, los datos del conjunto fueron generadors a partir de simulaciones de Monte Carlo.

4.3. Experimentos para evaluar el mapa autoorganizado.

4.3.1. Verificación de la implementación del modelo.

En el caso del mapa autoorganizado tanto la versión como para CPU como para GPU ejecutan el mismo algoritmo por lo que la métrica de interés durante las ejecuciones realizadas es el tiempo de ejecución. En primer lugar, durante la fase de desarrollo usamos el conjunto de las caras de *Olivetti*, que nos permitió comprobar de manera empírica y gráfica que los resultados son correctos. En caso de funcionar correctamente, generamos un conjunto de imagenes con la misma dimensión del mapa que son o se parecen a algunas de las caras de los sujetos y donde las imágenes más parecidas se encuentran próximas las unas con las otras.

Para evaluar que tanto nuestra implementación para CPU usando *NumPy* como nuestra implementación para CUDA son correctas generamos un mapa de 5 filas y 6 columnas y ejecutamos el algoritmo durante 50 iteraciones, con 25 para la primera fase y otras 25 para la segunda fase y con los parámetros de control σ_0, σ_f y τ a 3, 0,1 y 50, respectivamente. Además, puesto que aunque ambos algoritmos hagan los mismo utilizan métodos para la generación de los pesos aleatorios iniciales distintos, tomamos las dos medidas de calidad del mapa autoorganizado consideradas: el error de cuantificación y el error topográfico.

Figura 4.1: Imagen obtenida en el experimento para CPU del mapa autoorganizado.

Figura 4.2: Imagen obtenida en el experimento para GPU del mapa autoorganizado.

En la figura 4.1, podemos observar los resultados obtenidos para la ejecución de este algoritmo sobre CPU. En ella podemos observar como personas con piel de color más oscuro se encuentran hacia la esquina inferior izquierda o personas con gafas en la esquina superior derecha. En este ejemplo obtenemos un error de cuantificación de 6,55 y un error topográfico de 0,25, tardando un total de 251,49 segundos en su ejecución.

En la figura 4.2, podemos observar los resultados obtenidos para la ejecución de este algoritmo sobre nuestro dispositivo *CUDA*. En ella podemos observar como personas con piel de color más oscuro se encuentran hacia la esquina inferior derecha y todas las personas de la derecha llevan gafas, entre otros detalles. Para este ejemplo obtenemos un error de cuantificación de 6,57 y un error topográfico de 0.02, tardando un total de 357,41 segundos en su ejecución.

En este pequeño experimento hemos podido comprobar visualmente que ambas implementaciones funcionan correctamente y proporcionan resultados similares. Gran parte de los errores de implementación que ocurrireron durante el desarrollo de este algoritmo del proyecto fueron detectados gracias a este experimento. El hecho de que la versión para GPU tarde más que la versión para CPU se debe al reducido número de muestras de este conjunto de datos (400) que no llegan a utilizar ni la mitad de la capacidad de nuestra tarjeta gráfica.

4.3.2. Uso del modelo sobre un conjunto de datos grandes dimensiones.

Posteriormente, para evaluar la capacidad del algoritmo ante un conjunto de mayores dimensiones, utilizamos SUSY. Para este experimento ignoramos las etiquetas de salida y utilizamos para un mapa de neuronas de 8 filas y 7 columnas con los parámetros de control τ a 10, σ_0 a 4, σ_f a 0,1. El algoritmo lo ejecutamos durante 10 iteraciones (5 cada fase) y realizamos 5 repeticiones de cada experimento para tomar una medida de tiempo promedio, con el fin de obtener resultados más fiables que realizando una única ejecución. En este experimento nos centramos en evaluar como varía el rendimiento de nuestra implementación según vamos aumentando el número de muestras que evaluamos. Para ello, tomamos subcojuntos del RDD partición del 10 % del número de muestras del conjunto de datos (medio millón de muestras) y aumentado de 10 % en 10 % hasta evaluar el conjunto completo (5 millones de muestras).

Nº de Muestras	***Tiempo CPU (s)***	***Tiempo GPU (s)***	***Ganancia***
500000	1607,32	1024,68	1,57
1000000	2473,72	1038,77	2,38
1500000	3330,50	1068,09	3,12
2000000	4190,89	1092,70	3,84
2500000	5076,56	1119,72	4,53
3000000	5860,05	1137,68	5,15
3500000	6806,87	1158,58	5,88
4000000	7684,52	1171,87	6,56
4500000	8526,80	1181,42	7,22
5000000	9406,62	1197,52	7,80

Cuadro 4.1: Tiempos promedios de ejecución y ganancias para el experimento del mapa autoorganizado sobre SUSY.

En la tabla 4.1 vemos las diferencias entre los tiempos promedios de 5 ejecuciones para CPU y 5 ejecuciones para GPU según los percentiles de muestras propuestos para el experimento. La evolución de los tiempos de ejecución para la GPU oscila en un pequeño intervalo entre los 1025 segundos (17 minutos) y casi 2000 segundos (20 minutos). Sin embargo, la evolución de los tiempos para la CPU osicla entre los 1607 segundos (27 minutos) y los 9406 minutos (2 horas y 37 minutos). Para una mejor visualización de estos resultados planteamos la gráfica de la figura 4.3, en la que combinamos las gráficas de lineas para la evolución de los tiempos promedios con las ganancias obtenidas en una gráfica de barras.

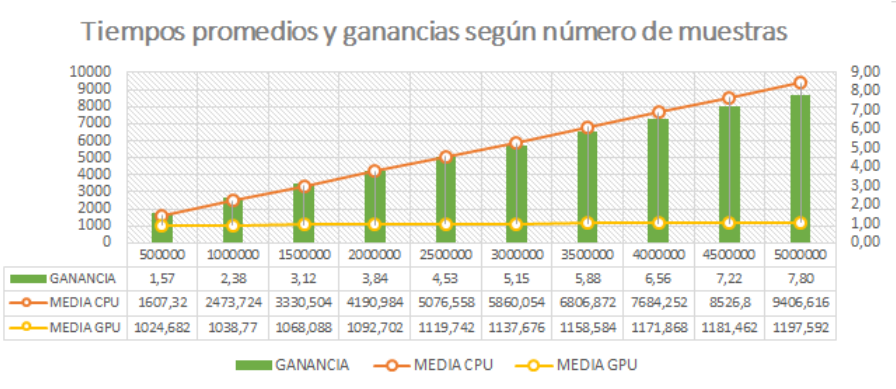


Figura 4.3: Gráfica con tiempos promedios y ganancias para SUSY.

En la gráfica planteada vemos de manera clara cómo al aumentar el número de muestras nuestra implementación que combina *CUDA* y *Spark* rinde considerablemente mejor. Antes de resaltar algunos de los resultados obtenidos, hemos de tener en cuenta que, al leer el archivo CSV, *Spark* generó un *RDD* con 72 particiones, es decir, 6 particiones para cada uno de los 12 núcleos (6 físicos y 6 lógicos) de nuestro AMD Ryzen 5 2600X. A la hora de evaluar con la GPU hemos mantenido el mismo número de particiones, por lo que en al evaluar 500000 en realidad estamos realizando 72 ejecuciones independientes del algoritmo y combinando los resutados obtenidos de todas esas particiones. Mientras que la generación de múltiples particiones facilita que los conflictos que se puedan generar para realizar las operaciones atómicas sean menores, resulta claro que conforme aumentamos el tamaño de la muestra obtenemos ganancias considerablemente mayores, llegando a tener un tiempo de ejecución casi 8 veces más rápido en nuestra GTX 1060 que en los 12 cores de nuestro Ryzen para el ejemplo más complejo que hemos planteado.

4.3.3. Resultados de nvprof sobre la versión final del algoritmo.

Por último, mostramos los resultados obtenidos de un ejemplo de ejecución que realiza los cálculos parciales para generar un mapa de 10x10 neuronas sobre un problema de dimensión 18 y con 100000 muestras y con el parámetro de control $\sigma^2 = 10$. Los resultados obtenidos por el mismo los hemos sintetizado en el siguiente diagrama de sectores, donde cada sector representa el porcentaje del tiempo total que cada actividad de la GPU es usado del tiempo total de ejecución.

En la figura 4.4 vemos el diagrama de sectores en cuestión, que nos

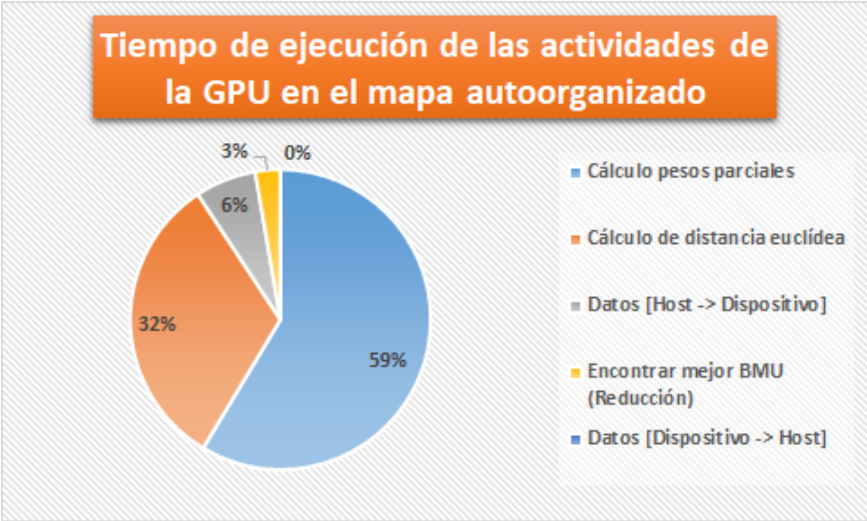


Figura 4.4: Diagrama de sectores de los resultados de nvprof para el mapa autoorganizado.

indica que para una ejecución el cuello de botella es el *kernel* que realiza los cálculos parciales de numeradores y denominadores de la fórmula de la actualización de pesos, ocupando un 59 % de tiempo de ejecución seguido del cálculo de la distancia euclídea con un 32 % del tiempo de ejecución, es decir, el segundo *kernel* es prácticamente el doble de rápido que el primero. Estas dos operaciones ocupan alrededor del 90 % del tiempoo de ejecución, dejando tan sólo un 3 % del tiempo para la reducción, un 6 % en transladar las muestras al dispositivo y una cantidad de tiempo despreciable en devolver los resultados obtenidos al *host*. Básandonos en los resultados obtenidos, hemos de hacer hincapié en que posibles futuros trabajos basados en la implementación del mapa autoorganizado que hemos desarrollado han de centrarse principalmente en buscar mejores maneras de realizar esos cálculos parciales que requerían de operaciones atómicas o, en su defecto, mejorar el cálculo de la distancia euclídea o incluso utilizar otras medidas de distancia que puedan ser paralelizadas con mayor eficiencia.

4.4. Experimentos para evaluar el árbol de decisión.

Para los experimentos relaciones con estos problemas de clasificación tenemos en cuenta dos métricas: el tiempo de ejecución y la precisión (porcentaje de predicciones correctas).

En primer lugar, durante el proceso de desarrollo, se empezó a evaluar la creación de un único árbol con profundidad limitada entre la implementación desarrollada y la implementación de Spark. Hemos de destacar, antes de comentar los resultados obtenidos, que, a diferencia del modelo anterior, el algoritmo utilizado para el cálculo en la CPU y el creado en CUDA no hacen lo mismo (el nuestro esta basado en una búsqueda exhaustiva de puntos de corte y el de Spark en una discretización por cuantiles y el uso de histogramas) aunque ambos generen un árbol de decisión. Los resultados obtenidos fueron generados mediante un proceso de validación cruzada *leave-one out* con 10 particiones, es decir se generaron 10 particiones aleatorias de los datos y, en cada iteración, una es seleccionada para comprobar la precisión del modelo obtenido y el resto para entrenarlo, de tal manera que todas las particiones son utilizada para comprobar la calidad del modelo generado por el resto de ellas. Los resultados obtenidos de la validación cruzada son el promedio de estas iteraciones.

4.4.1. Tablas de resultados para la generación de un único árbol.

	SPAMBASE (4601 muestras, 57 atributos)			
	CUDA		*SPARK*	
Profundidad	*Tiempo (s)*	*Precisión (%)*	*Tiempo (s)*	*Precisión (%)*
4	**2,58**	88,13	6,47	**88,64**
5	**2,92**	85,7	6,48	**90,77**
6	**4,44**	86,2	6,6	**91,14**
7	**7,32**	89,72	6,75	**91,9**
8	10,81	90,15	**6,84**	**91,95**
9	12,15	85,8	**6,9**	**92,29**

Cuadro 4.2: Validación cruzada para árbol de decisión en spambase.

	MAGIC (19020 muestras, 11 atributos)			
	CUDA		*SPARK*	
Profundidad	*Tiempo (s)*	*Precisión (%)*	*Tiempo (s)*	*Precisión (%)*
4	**0,36**	79,37	6,42	**81,41**
5	**0,65**	80,99	6,48	**81,71**
6	**1,17**	81,77	6,58	**83,59**
7	**2,09**	83,87	6,71	**84,23**
8	**3,52**	84,1	6,81	**84,55**
9	**5,77**	84,2	7,05	**84,73**

Cuadro 4.3: Validación cruzada para árbol de decisión en MAGIC.

4.4.2. Análisis de los resultados para la generación de un único árbol.

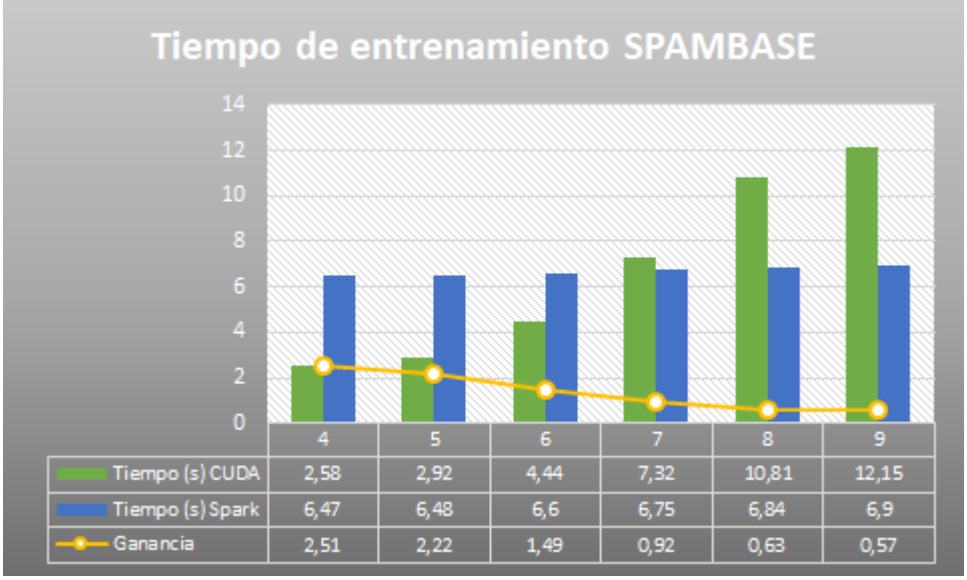
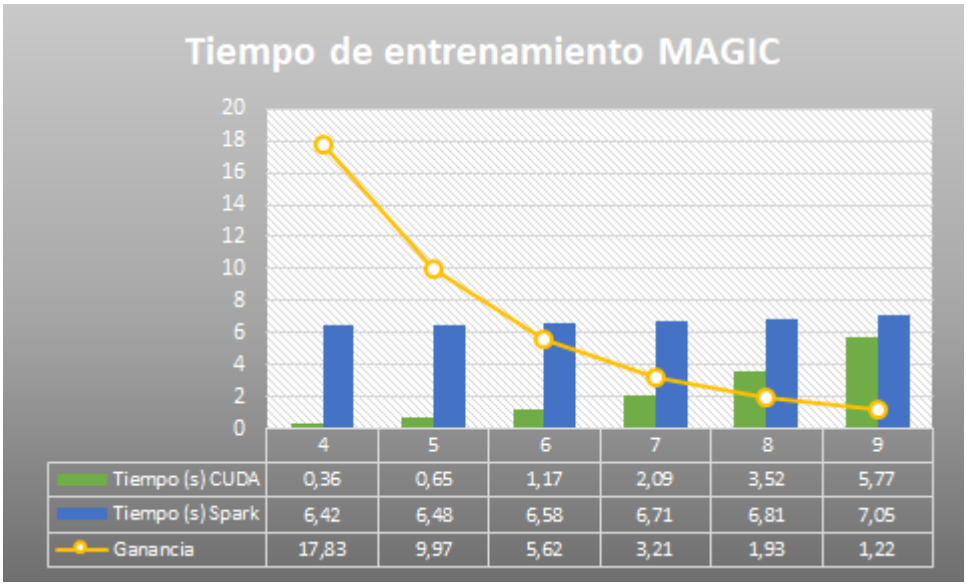


Figura 4.5: Tiempo de entrenamiento y ganacia según profundidad en Spambase.

En la figura 4.5 podemos observar cómo evolucionan los tiempos de ejecución del modelo para CPU (en azul) y CUDA (en verde) según vamos aumentando la profundidad de los árboles generados empezando en cuarto nivel de profundidad y terminando en el noveno. Podemos concluir con facilidad que el rendimiento de la implementación en CUDA empeora considerablemente conforme aumentamos los niveles de profundidad, incluso llegando a ser más lento que la versión en CPU de *Spark*, factor que resulta razonable pues, conforme vamos aumentando dicho nivel más kernels han de ser lanzados y el número de muestras en cada uno va a ser inferior pudiendo llegar a ser incapaz de aprovechar la capacidad de procesamiento que tienen los núcleos de CUDA por ejemplo sin el nodo no hay suficientes elementos para utilizar todas las hebras de un bloque o siquiera las de un *warp*, así como la reorganización de los datos en la memoria que sigue siendo una operación bastante costosa, aunque no haga falta aplicar de nuevo un algoritmo de ordenación, que ocurre en cada nivel.

La generación de un árbol de decisión para Spambase expone los dos problemas principales cuando utilizamos este algoritmo: tener un número de muestras relativamente reducido y una gran cantidad de listas de atributos. Si tenemos un número de muestras más o menos reducido, en este caso

tenemos 4600 muestras, el nivel de profundidad én el que la utilización de la GPU de manera efectiva decae considerablemente se alcanza antes. Por otro lado, el elevado número de atributos (56 más la etiqueta de salida) implica la generación de 56 listas de atributos que generan tanto un *overhead* en la complejidad espacial del modelo en la GPU como en la cantidad de transferencias de datos que se han de realizar en la memoria global del dispositivo CUDA.



	4	5	6	7	8	9
Tiempo (s) CUDA	0,36	0,65	1,17	2,09	3,52	5,77
Tiempo (s) Spark	6,42	6,48	6,58	6,71	6,81	7,05
Ganancia	17,83	9,97	5,62	3,21	1,93	1,22

Figura 4.6: Tiempo de entrenamiento y ganacia según profundidad en MAGIC.

En la figura 4.6 realizamos el mismo análisis para el conjunto de datos *MAGIC Gamma Telescope*. En este caso nos encontramos, en cierta manera, con la situación contraria a la que observábamos en Spambase, el número de muestras considerablemente más elevado (19020) y el número de atributos mucho más reducido (10 más la etiqueta de salida) ponen de manifiesto la velocidad de la ejecución del algoritmo cuando se dan las condiciones ideales para su uso. Cabe destacar que para profundidades muy reducidas, como ocurre en el nivel de profundidad 4, obtenemos una gran ganancia, siendo 17 veces más rápida nuestra implementación que la de *Spark* y manteniendo ganancias considerables para profundidades 6 y 7 (5,62 y 3,21, respectivamente).

Puesto que, como comentábamos previamente, ambas implementaciones no hacen exactamente lo mismo es importante observar también las diferencias existentes en términos de precisión para ambos modelos.

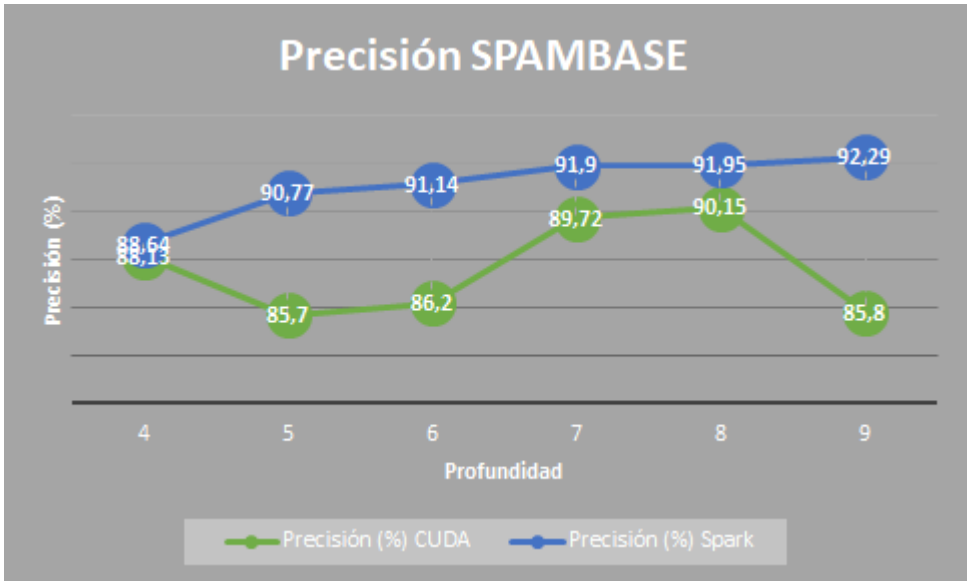


Figura 4.7: Precisión según profundidad en SPAMBASE.

La precisión de nuestra implementación según la profundidad (figura 4.7) es considerablemente inferior en la mayoría de niveles de profundidad para el entrenamiento de *Spambase*. La posibilidad de que muestras ruidosas afecten a la calidad de los resultados obtenidos en nuestro modelo es mucho mayor que en el de *Spark* pues no hemos utilizado ninguna técnica de poda más avanzada que limitar la profundidad del árbol generado. Dependiendo del nivel de profundidad, nuestra implementación llega a un rango competitivo de precisión con *Spark* o queda considerablemente por detrás, siendo el caso más claro la profundidad 9 donde *Spark* obtiene una ventaja en precisión del 6,49 %.

Por otro lado, en la figura 4.8, podemos observar que no sólo podemos tener problemas por la falta de técnicas de poda avanzadas sino que puesto que ambos algoritmos generan un árbol de decisión de maneras considerablemente distintas habrá situaciones en las que las diferencias entre uno y otro sean mayores y otras en las que sean ambas muy competentes, como en este caso que según el nivel de profundidad la diferencia de precisión varía entre el 0,36 % y el 2,04 %.

Una vez vistos los puntos fuertes y débiles de nuestra implementación a la hora de probar con nuestro conjunto de *Big Data*, *SUSY*, decidimos en vez de generar un único árbol generar un *random forest* con nuestros árboles de profundidad limitada. En concreto, para el entrenamiento tene-

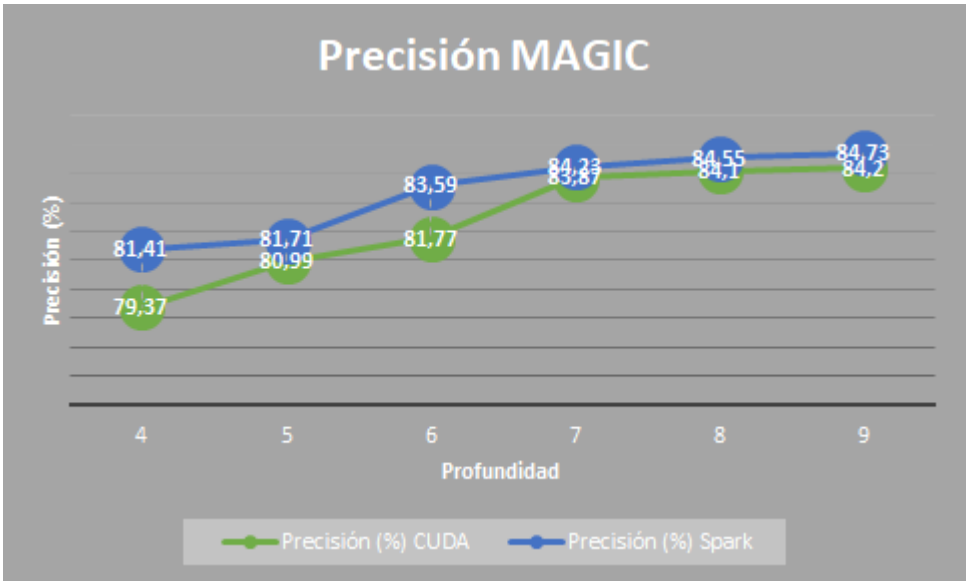


Figura 4.8: Precisión según profundidad en MAGIC.

mos 4 millones y medio de muestras de 18 atributos más la etiqueta de clasificación. Para nuestro experimento sobre este conjunto hemos generado un *random forest* de 225 árboles utilizando cada uno de ellos 20000 de la muestras presentes en el conjunto y utilizando todos los atributos para entrenar cada árbol. Hemos realizado la misma validación cruzada y realizado el experimento para el sexto y el séptimo nivel de profundidad.

4.4.3. Tabla de resultados del random forest.

	CUDA		*SPARK*	
Repetición	***Tiempo (s)***	***Precisión (%)***	***Tiempo (s)***	***Precisión (%)***
1	**560,48**	78,12	1201,04	**85,3**
2	**536,01**	78,14	1273,04	**85,3**
3	**519,96**	78,24	1236,47	**85,38**
4	**519,47**	78,15	1238,79	**85,33**
5	**530,59**	78,13	1334,8	**85,33**
6	**543,35**	78,11	1324,34	**85,24**
7	**545,54**	78,15	1187,43	**85,27**
8	**530,74**	78,1	1247,09	**85,29**
9	**558,46**	78,18	1251	**85,34**
10	**521,29**	78,1	1268,34	**85,28**
MEDIA	***536,59***	***78,14***	***1256,23***	***85,31***

Cuadro 4.4: Resultados de validación cruzada en SUSY para profundidad 6.

	CUDA		*SPARK*	
Repetición	***Tiempo (s)***	***Precisión (%)***	***Tiempo (s)***	***Precisión (%)***
1	**583,13**	78,84	1563,44	**85,78**
2	**559,27**	78,83	1692,27	**85,8**
3	**584,07**	78,97	1768,84	**85,88**
4	**548,74**	78,86	1752,11	**85,82**
5	**561,02**	78,78	1681,44	**85,82**
6	**574,28**	78,76	1734,05	**85,72**
7	**572,08**	78,84	1790,56	**85,77**
8	**580,59**	78,81	1769,53	**85,76**
9	**576,32**	78,81	1816,56	**85,82**
10	**592,13**	78,77	1175,02	**85,76**
MEDIA	***573,16***	***78,83***	***1674,38***	***85,79***

Cuadro 4.5: Resultados de validación cruzada en SUSY para profundidad 7.

4.4.4. Análisis de los resultados del random forest.

A la hora de realizar este experimento podemos observar cómo los tiempos de ejecución de cada iteración se han disparado para afrontar el problema de mayores dimensiones. La generación de múltiples árboles pequeños nos ha llevado a tardar un promedio de 536,59 segundos para profundidad 6 y 573,16 segundos, es decir un periodo de entre 9 y 10 minutos utilizando la GPU. En el caso de la versión de *Spark* para CPU, en profundidad 6 hemos tardado 1256,23 segundos (unos 20 minutos) y 1674,38 segundos para profundiad 7 (casi 28 minutos), dando lugar a una ganancia promedio de 2,34 para profundidad 6 y de 2,92 para profundidad 7. Mientras que en términos de velocidad de entrenamiento nuestro algoritmo ha sido considerablemente superior incluso incrementando la ganancia al pasar de un nivel de profundidad a otro, hemos de tener en cuenta también la diferencia existente en términos de precisión, que deja nuestra implementación un 7,16 % de precisión peor para profundidad 6 y un 6,97 % peor para profundidad 7.

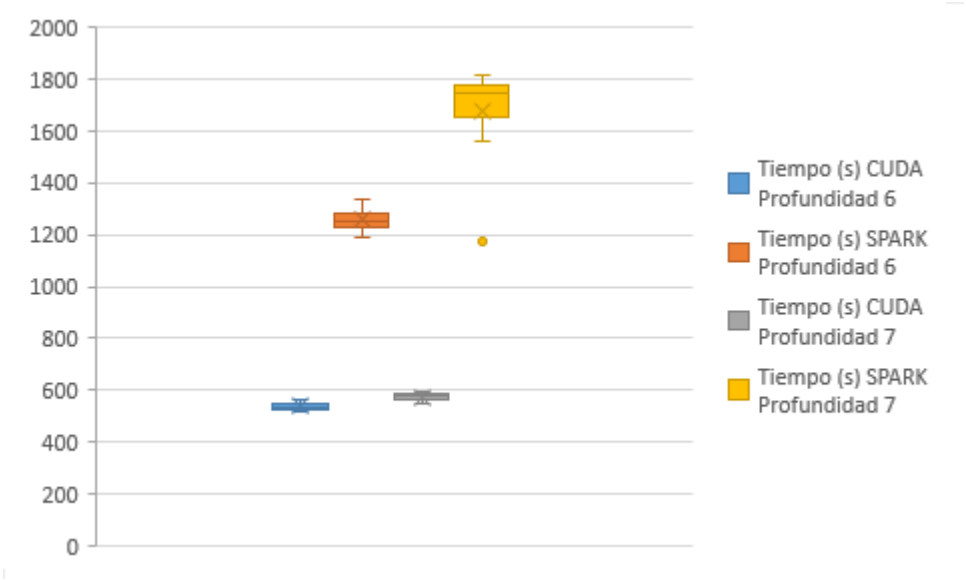


Figura 4.9: Diagrama de cajas y bigotes para el tiempo de entrenamiento del random forest para SUSY.

De todos los experimentos realizados podemos concluir que mientras que, efectivamente nuestro modelo es más rápido siempre y cuando no generemos árboles demasiado profundos o tengamos una cantidad de atributos muy elevada. En este trabajo, nos hemos centrado en conseguir esa ganancia durante el proceso de entrenamiento. No obstante, pueden ser líneas de trabajo futuras sobre este modelo un análisis específico de los parámetros de

profundidad y número de árboles óptimos, la utilización de otros criterios además del utilizado, o el uso del tiempo extra obtenido para mejorar los resultados ya sea con algún preprocesamiento de datos o el uso de criterios de poda más avanzados, entre otros.

4.4.5. Resultados de nvprof sobre la versión final del algoritmo.

Por último, como realizábamos para el modelo anterior, analizamos el punto final en el que hemos dejado nuestra implementación utilizando el profiler de NVIDIA, *nvprof*. Para realizar el *profile*, entrenamos MAGIC en un árbol de profundidad 6. El diagrama de sectores de la figura 4.10 nos presenta un resumen de dichos resultados.

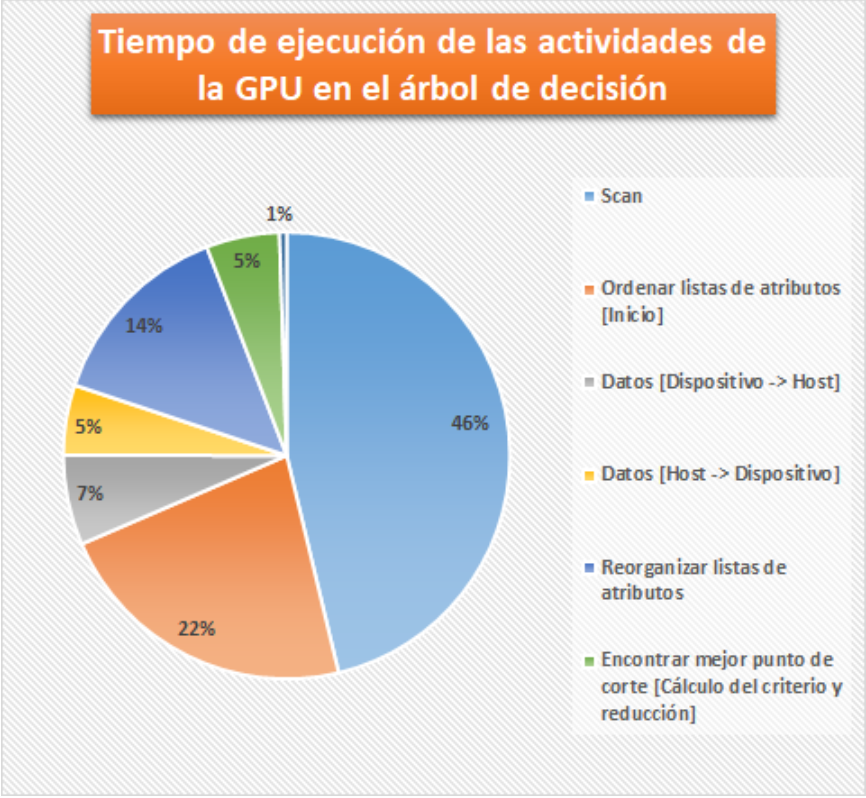


Figura 4.10: Diagrama de sectores de los resultados de nvprof para el árbol de decision.

Como cabía esperar el cuello de botella es la implementación de la primitiva de *scan* pues es utilizada amplia y reiteradamente durante la ejecución

del algoritmo con un 46 % seguido de la implementación del *Radix Sort* de *CuPy* que utilizamos para organizar las listas de atributos al inicio del algoritmo con un 22 % y posteriormente la reorganización de las listas de atributos con un 14 %. Dados estos resultados, queda claro que, si es posible, es prioritario optimizar la operación de *scan* implementada aunque buscar alternativas para la organización inicial de las listas de atributos o la reorganización que ocurre cada nivel serían también opciones interesantes.

Capítulo 5

Conclusiones y trabajos futuros.

Bibliografía

[1] NVIDIA, "Procesamiento paralelo cuda," consultado el 27 de abril de 2019. [Online]. Available: https://www.nvidia.es/object/cuda-parallel-computing-es.html

[2] T. Kohonen, "The self-organizing map," *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1464–1480, Sep. 1990.

[3] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*, ser. Statistics/Probability Series. Belmont, California, U.S.A.: Wadsworth Publishing Company, 1984.

[4] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/2934664

[5] F. Codevilla, S. Botelho, N. Duarte Filho, and J. Gaya, "Parallel high dimensional self organizing maps using cuda," 10 2012, pp. 302–306.

[6] H. Daneshpajouh, P. Delisle, J.-C. Boisson, M. Krajecki, and N. Zakaria, *Parallel Batch Self-Organizing Map on Graphics Processing Unit Using CUDA*, 01 2018, pp. 87–100.

[7] W.-T. Lo, Y.-S. Chang, R.-K. Sheu, C.-C. Chiu, and S.-M. Yuan, "Cudt: a cuda based decision tree algorithm," *TheScientificWorldJournal*, vol. 2014, p. 745640, 07 2014.

[8] D. Svatensson, "Implementing streaming parallel decision trees on graphic processing units," 06 2018. [Online]. Available: http://www.diva-portal.se/smash/get/diva2:1220512/FULLTEXT02.pdf

[9] J.-C. Fort, P. Letrémy, and M. Cottrell, "Advantages and drawbacks of the batch kohonen algorithm," 01 2002, pp. 223–230.

[10] F. C. y. M. J. M. B. Fernando Berzal, Juan Carlos Cubero, "On the quest for easy to understand splitting rules," *Data & Knowledge Engineering*, vol. 44, no. 1, pp. 31 – 48, 2003.

[11] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[12] S. Kwan Lam, A. Pitrou, and S. Seibert, "Numba: a llvm-based python jit compiler," 11 2015, pp. 1–6.

[13] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "Cupy: A numpy-compatible library for nvidia gpu calculations," in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017. [Online]. Available: http://learningsys.org/nips17/assets/papers/paper_16.pdf

[14] M. Harris, "Optimizing parallel reduction in cuda," *Proc. ACM SIGMOD*, vol. 21, pp. 104–110, 01 2007.

[15] J. Shafer, R. Agrawal, and M. Mehta, "Sprint: A scalable parallel classifier for data mining," *VLDB*, 08 2000.

[16] S. Sengupta, M. Harris, M. Garland, and J. Owens, *Efficient Parallel Scan Algorithms for GPUs*, 01 2011, pp. 413–442.

[17] d. d. S.-L. AT&T Laboratories Cambridge, "The olivetti faces dataset," 1994, enlace consultado el 6 de abril de 2019. [Online]. Available: https://scikit-learn.org/0.19/datasets/olivetti_faces.html

[18] D. Dua and C. Graff, "Spambase, UCI machine learning repository," consultado el 20 de abril de 2019. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/spambase

[19] ——, "Magic, UCI machine learning repository," consultado el 20 de abril de 2019. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/magic+gamma+telescope

[20] P. Baldi, P. Sadowski, and D. Whiteson, "Searching for exotic particles in high-energy physics with deep learning," *Nature communications*, vol. 5, p. 4308, 07 2014.