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, septiembre de 2019

Desarrollo e Implementación de modelos paralelos de Soft Computing en CUDA.

David Criado Ramón

Palabras clave: Soft Computing, CUDA, GPU, paralelo, mapa auto-organizado, Spark, Kohonen, árbol de decisión, CART, CUDT, Python, Numba, SUSY, reducción, scan.

Resumen

El objetivo de este proyecto es la paralelización de varios algoritmos de *Soft Computing* usando la tecnología propietaria de *NVIDIA* para sus tarjetas gráficas, *CUDA*. Dos algoritmos fueron seleccionados para su desarrollo: el mapa auto-organizado de *Kohonen* y el árbol de decisión *CART*. Además, para simplificar el proceso de desarrollo nos apoyamos en el *framework* para computación en clúster *Apache Spark*, que nos ha permitido desarrollar una solución viable para ser utilizada bien en una única máquina con un dispositivo *CUDA* o en un clúster con múltiples máquinas, cada una con su dispositivo CUDA correspondiente.

En el caso del mapa auto-organizado de Kohonen planteamos una solución basada en la primitiva paralela de la reducción y el uso de operaciones atómicas. Además, para evitar *overheads* debidos al lanzamiento de múltiples *kernels*, limitamos el tamaño del mapa de salida a 1024 neuronas.

En el caso del árbol de decisión *CART*, seguimos una alternativa similar a CUDT, basándonos en la primitiva paralela de suma acumulada o *scan* y limitando los problemas a resolver a clasificación binaria, añadiendo algunas técnicas extra, como los *streams*, para obtener resultados ligeramente mejores en tiempo de ejecución.

Para comprobar el rendimiento de ambos modelos hemos utilizado la base de datos *SUSY*, compuesta por 5 millones de muestras con 18 características y una clase binaria correspondiente y, haciendo uso de *NumPy*, hemos desarrollado las versiones equivalentes utilizando la CPU. En las pruebas realizadas llegamos a obtener un tiempo de ejecución hasta 28 más rápido para el mapa auto-organizado de *Kohonen* con el dispositivo *CUDA*. Sin embargo, para el árbol de decisión desarrollado, aunque conseguimos tiempos de ejecución más rápidos, obtenemos resultados mucho más moderados, con tiempos de ejecución hasta 1,5 veces más rápidos.

Development and Implementation of Soft Computing parallel models using CUDA.

David Criado Ramón

Keywords: Soft Computing, CUDA, GPU, parallel, self-organizing map, Spark, Kohonen, decision tree, CART, CUDT, Python, Numba, SUSY, reduction, scan.

Abstract

In this project, we parallelize two Soft Computing models using CUDA: self-organizing maps, a competitive and unsupervised learning neuronal network and CART decision trees, an exhaustive search decision tree algorithm suitable for classification and regression tasks (though we are limiting here our implementation, based on GPU, to binary classification problems).

Both models are developed using Python and *Apache Spark*. *Python* allows us to implement our code faster via *Numba* than using the more traditional approach with*C++* and *CUDA* and, thereby, it offers an easy integration with Spark. On the other hand, *Spark* offers an easy way to read CSV files, manageable parameters to solve scalability issues (executors, number of cores, étc.) and the capability to carry out only one implementation, which is suitable for execution either on one machine. In order to compare GPU performance vs CPU, we use NumPY and Spark to create CPU-based implementations that work like the GPU ones.

The self-organizing map (SOM), proposed by Kohonen in the early 80s, is an unsupervised learning algorithm suitable for clustering and dimensionality reduction, among others. Our solution, based on the batch version of the SOM, uses Spark's *mapPartition* transformation in order to distribute work among all the partitions of the RDD. Each iteration, on each partition, our implementation uses the Euclidean distance to calculate distance between the neurons, as well as the parallel reduction primitive in order to find the closest neuron (BMU) and updates the weights structure using atomic operations. Furthermore, we limit the number of neurons on the output map to 1024 in order to avoid overheads by multiple kernel launches. We achieve an speedup of almost 28 times compared with the CPU implementation on the selected dataset, SUSY, composed by 5 million instances with 18 features.

The development of a decision tree learning algorithm proved to be much

more challenging. Being based on CART, we decided to develop a solution similar to CUDT, some small tweaks like common on-line pruning techniques: changing depth and number of samples assigned to the leaves of trees and the use of streams to allow concurrent evaluation of nodes in the GPU. In this algorithm implementation, by using Python limits the implementation of the algorithm since Numba does not support CUDA dynamic parallelism, thenforcing us to launch multiple kernels instead. Also, being based on CUDT, our implementation only solves binary classification problems and uses the parallel scan primitive, which we implemented using warp constraint. We incorporate Spark to this model by creating a random forest of these trees. This will help to prevent overfitting and will be, in fact, faster than using Spark to create a single tree, since the latter would create a serious communication overhead between the RDD's partitions. To test our implementation, we use SUSY dataset (5 million samples, 18 features, 1 binary class label). Our GPU implementation was able to improve the CPU one, however, the best speedup we were able to obtain in our system was only 1.5 times faster than the second one, although parameters like the maximum depth, the number of trees in the random forest or the number of samples of the dataset may have heavily impacted the results obtained.

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 3 de septiembre 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 3 de septiembre de 2019.

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. Introducción y 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, su uso es muy popular en la comunidad científica.

La estructura de la GPU *(Graphics Processing Unit)*, utilizando un mayor número de núcleos a cambio de una velocidad de reloj más baja a la que podemos encontrar en una CPU *(Central Processing Unit)*, es de especial utilidad en operaciones masivamente paralelas, pudiendo llegar a proporcionar tiempos de ejecución considerablemente mejores a los que podríamos obtener usando una CPU.

Por otro lado, los algoritmos y técnicas de *Soft Computing* se corresponden 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*, combinando *CUDA* con el *framework* de computación en clúster *Spark* [2]. De esta manera, los algoritmos que se desarrollen podrán ser ejecutados tanto en un único dispositivo como en un clúster con múltiples dispositivos *CUDA*. Para ello, se analizarán los algoritmos y sus posibilidades de paralelización, se realizarán las implementaciones adecuadas y se evaluará el rendimiento de las mismas

utilizando conjunto de datos con un número de muestras elevado.

Tras evaluar varias opciones, se optó por desarrollar dos modelos distintos: los mapa auto-organizados de Kohonen [3] y los árboles de decisión [4].

1.2. Descripción del proyecto.

En este proyecto se pretende que el alumno diseñe, desarrolle e implemente modelos en paralelo asociados a algoritmos tradicionales de *Soft Somputing*. Para ello se utilizará el lenguaje *CUDA*, pudiendo de esta manera aprovechar las características de los dispositivos GPUs. Para probar estos modelos se escogerán problemas relacionados con Big Data y que tengan una gran carga computacional.

1.3. Requisitos de hardware y software para el proyecto.

Requisitos de *hardware*
El único requisito de *hardware* en este proyecto es disponer de un sistema con un dispositivo *CUDA*.

Dependencias de *software*
Para las implementaciones del proyecto se han usado:

- Los *drivers* apropiados para el dispositivo *CUDA* del sistema.
- *Python 3.6* con los paquetes *NumPy* y *Numba*.
- *Spark* 2.4.0 con *Hadoop* 2.7.3

1.4. Planificación de tareas.

Este proyecto fue desarrollado mayoritariamente durante el segundo cuatrimestre del curso, precedido por el estudio de las tecnologías utilizadas (*CUDA* y *Spark*).

Tarea	***Fecha finalización***
Estudio sobre CUDA.	15 Marzo 2019
Estudio de bibliografía y selección de modelo a implementar: Árbol CART.	28 Marzo 2019
Estudio de bibliografía del modelo: Mapa auto-organizado on-line.	7 Abril 2019
Estudio de bibliografía y selcción de modelo a implementar: Mapa auto-organizado batch	12 Abril 2019
Implementación en CUDA de mapa auto-organizado batch.	17 Abril 2019
Implementación en CUDA de árbol de decisión CART (sólo clasificación binaria).	23 Abril 2019
Estudio de Spark.	6 Mayo 2019
Incorporación de Spark al mapa auto-organizado batch	9 Mayo 2019
Incorporación de Spark al árbol de decisión.	14 Mayo 2019
Primera fase de redacción de memoria.	25 Mayo 2019
Revisión y corrección de erratas en mapa auto-organizado.	5 Junio 2019
Revisión y optimización del árbol de decisión.	29 Junio 2019
Finalización de la memoria.	30 Julio 2019

Cuadro 1.1: Tareas realizadas durante el desarrollo del proyecto.

1.5. 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.

1.6. Estructura del documento.

- En el primer capítulo, **Introducción y motivación**, hemos comentado los propósitos 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, **Tecnologías utilizadas**, hacemos una introducción al uso de CUDA y Spark para los propósitos del documento.
- En el cuarto capítulo, **Estado del arte: trabajos relacionados**, realizamos un breve repaso de algunas implementaciones de los modelos propuestos que hace uso de CUDA presentes en la literatura.
- En el quinto 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 sexto 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 ampliar nuestro trabajo.

Capítulo 2

Modelos de Soft Computing considerados.

2.1. Mapas auto-organizados *(Self Organizing Map)*

A principio de la década de los 80, el científico finlandés Teuvo Kohonen [3], planteó un modelo de aprendizaje automático no supervisado y competitivo basándose en el estudio del funcionamiento del córtex cerebral. El modelo planteado, denominado mapa auto-organizado, red auto-organizada o red neuronal de Kohonen, entre otros nombres similares, es una red neuronal artificial, y las principales características que la definen son las siguientes:

- Es una **red neuronal artificial**. Esto quiere decir, a grandes rasgos, que la estructura que genera el modelo está basada en una red de múltiples unidades, llamadas "neuronas", que se encuentran interconectadas entre sí.
- La red neuronal de Kohonen tiene **dos capas**. Una capa de entrada, con tantas neuronas como características o atributos tenga el vector que representa a la muestra que vaya a ser evaluada por la red, es decir, tanto una muestra como la capa de entrada tendrán la misma dimensión y, una característica, podría ser, por ejemplo, la altura o el peso de una persona. La segunda capa es la capa competitiva o capa de Kohonen, de un tamaño a decidir por el usuario y una estructura habitualmente bidimensional, aunque podría perfectamente usarse cualquier otro número de dimensiones.
- Asociado a cada neurona de la capa competitiva, tenemos un vector de referencia variable $w_i(t) \in \mathbb{R}^n, i = 1, 2, ..., k$, es decir, un vector de "referencia" con respecto a la representación vectorial de las muestras

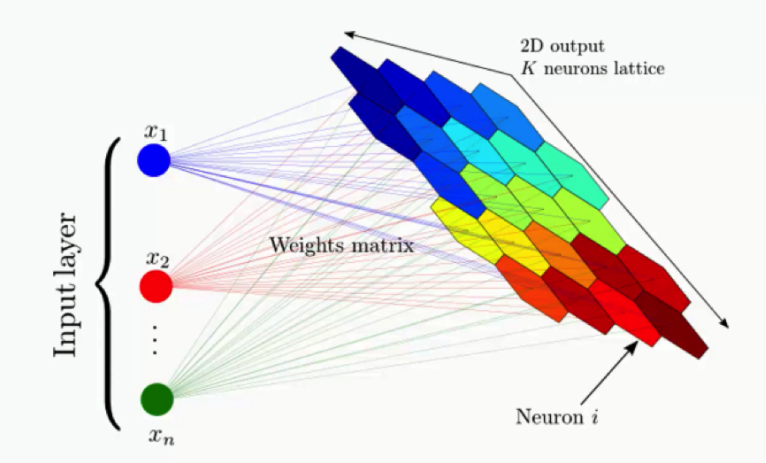

Figura 2.1: Esquema de una red neuronal de Kohonen.

estadísticas $\vec{X} = \vec{X}(t) \in \mathbb{R}^n$ que se corresponden al conjunto de datos de entrada para el entrenamiento del modelo. Estos vectores también son denominados vectores de pesos y, al conjunto de ellos, se le denomina *codebook*. En los casos en los que el algoritmo sea aplicado para realizar clustering, dicho vector representará el valor promedio de los vectores de características de las muestras asociadas al clúster de su neurona.

- Asociado a cada neurona de la capa competitiva, tenemos un vector de 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**, es decir, es 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.

Nota: Por simplicidad en la notación y en la explicación vamos a considerar que la capa competitiva es unidimensional. La explicación propuesta es extensible para un número de dimensiones arbitrario realizando los cálculos relacionados con la posición del vector de referencia dentro del codebook y

la posición de la BMU utilizando vectores con un valor para cada una de las dimensiones correspondientes.

Sea $\vec{X}$ un vector, que representa una secuencia de muestras estadísticas de un observable $x = x(t) \in \mathbb{R}$, donde t es el instante o iteración de la extracción de la muestra y un conjunto de vectores de referencia variables (*codebook*), $w_i(t)$, que representaremos de forma abreviada como el vector $\vec{W(t)} : w_i(t) \in \mathbb{R}\ |i : 0, 1, 2, ..., n$, donde los $w_i(0)$ están inicializados a un valor aleatorio siguiendo una distribución uniforme entre 0 y 1, se realizan los siguientes pasos, de forma iterativa, hasta alcanzar el número de iteraciones máximo λ, que determina el usuario como criterio de terminación para el proceso de entrenamiento.

En primer lugar, una muestra del vector $\vec{X}$ **será comparada simultáneamente con cada vector de referencia** $\vec{W(t)} = \{w_0(t), w_1(t), ..., w_i(t), ..., w_{n-1}(t)\}$ en cada instante de tiempo t, utilizando para ello la distancia euclídea entre vectores, $||X(t) - w_i(t)||_n$. Otras formulaciones para la distancia, como por ejemplo, la distancia de Mahalanobis, podrían haber sido usadas en su lugar.

A continuación, se determina **el vector de referencia más parecido a la muestra evaluada** en el instante t, y, a la ubicación que ocupa dicho vector de referencia (en índices) en el *codebook*, se le denomina **BMU *(Best Matching Unit)***.

$$BMU(X(t)) = argmin_{w \in \vec{W}} ||X - w||_n$$

La función *argmin* devuelve la posición del vector en la que se alcanza el valor mínimo y, a la distancia entre la muestra del instante t y su mejor vector de referencia la denominaremos

$$D(X(t)) = min_{w \in \vec{W}} ||X - w||_n$$

Una vez se ha seleccionado la BMU, se inicia un proceso de **actualización de los vectores de referencia** presentes en el vector $\vec{W}$. La obtención de un mejor ajuste para la siguiente iteración se realiza mediante la actualización del conjunto de vectores de referencia en función de la distancia entre la muestra y la BMU y la posición de la BMU en el vector $\vec{W}$. Este ajuste lo vamos a reflejar añadiendo a cada vector de referencia el término $\Delta W(t)$.

$$\vec{W}(t+1) = \vec{W}(t) + \Delta\vec{W}(t)$$

Durante este fase de actualización, tres parámetros de control pueden ser ajustados por el usuario. El parámetro de control τ tiene como objetivo marcar el ritmo al que los otros dos, el tamaño del vecindario y la tasa de aprendizaje, decrecen conforme avanza el tiempo siguiendo una función gaussiana. Es habitual utilizar, como τ, el número de iteraciones del proceso de entrenamiento. Por las características de los otros parámetros de control, que vamos a explicar a continuación, un menor valor de τ permite que el algoritmo realice una transición más rápida de una fase de exploración brusca, en la que gran parte de los vectores de referencia son modificados, a una fase de refinamiento en los que sólo el vector de referencia de la BMU y, quizás, los vectores más próximos, son ligeramente actualizados.

El tamaño del vecindario, determina un radio alrededor de la posición de la BMU en el vector $\vec{W(t)}$, en el que los vectores de referencias van a sufrir algún cambio. Si los vectores de referencia no se encuentran dentro del subconjunto $\vec{W}_A = \{w_{BMU(j)-\sigma^2}, ..., w_{BMU(j)}, ..., w_{BMU(j)+\sigma^2}\}$ su valor para $\delta_i(t)$ será 0. El parámetro σ es inicializado en el instante $t = 0$ por el usuario y actualizado conforme a una gaussiana. Además del sistema de decrecimiento exponencial presente en la gaussiana, posibilitamos que el usuario determine un valor fijo para σ, σ_t que puede establecerse a partir de una iteración determinada por el mismo, a la que denominamos z.

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

El tamaño del vecindario, σ, es utilizado para el cálculo de una función de vecindario, que en función del parámetro σ y la distancia (en índices) entre la BMU de cada vector de referencia $\vec{W(t)}$ pondera la modificación que va a ser realizada, permitiendo mantener propiedades topográficas y haciendo que, conforme nos alejamos (en índices) de la posición de la BMU en el vector $\vec{W}$ los cambios realizados sean menores. La función de vecindario, δ_f, viene dada por:

$$\delta_i(t) = e^{-\frac{||BMU(X(t))-i||}{2\sigma(t)^2}}.$$

En la figura 2.2, podemos observar una representación gráfica del vector $\vec{W}$ en un codebook bidimensional. Cada uno de los puntos rojos se corresponde con un vector de referencia, la BMU ha sido destacada de color amarillo y la región alrededor de la BMU que va a sufrir cambios significativos, el

vecindario, queda delimitada con círculo alrededor de la BMU cuyo radio es σ^2.

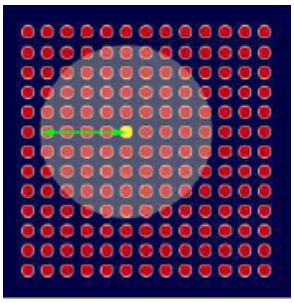

Figura 2.2: Representación gráfica del vecindario para un codebook bidimensional.

Por otro lado, la tasa de aprendizaje, η, tiene como objetivo ponderar la actualización de los pesos haciendo que, conforme avancen las iteraciones no sólo se vaya reduciendo el número de vectores de referencia que sufren cambios significativos, sino que, además, conforme avance el número de iteraciones incluso el vector de referencia de la BMU reciba cambios más ligeros. De manera similar al tamaño del vecindario, su valor inicial, η_0 es indicado por el usuario, y se puede establecer una iteración z en la que es fijado a un valor determinado.

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

Combinando la función de vecindario, la tasa de aprendizaje y la distancia entre la BMU y la muestra a ser evaluada en el instante t, obtenemos la fórmula que actualiza los vectores de referencia.

$$\Delta w_i(t) = \eta(t)\delta_i(t)(X(t) - w_i(t))$$

Una vez alcanzado el número de iteraciones máximo, el vector $\vec{W}$ contiene los vectores de referencia de la solución de tal manera que si, el algoritmo estuviese siendo usado para resolver problemas de clustering, cada vector de referencia representaría un clúster.

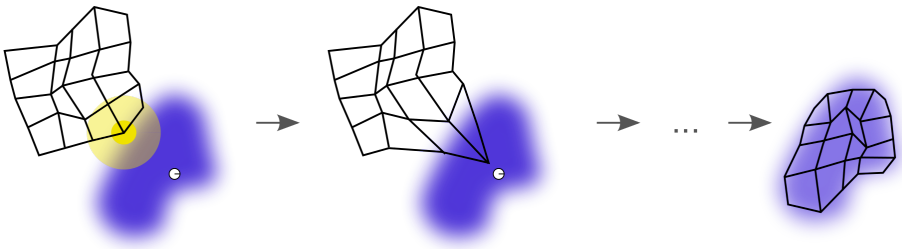

Figura 2.3: Proceso de entrenamiento de una red neuronal de Kohonen.

La figura 2.3 nos muestra, a grandes rasgos, un esquema del proceso de entrenamiento del mapa auto-organizado de Kohonen con una capa competitiva bidimensional.

En la primera parte de la figura, observamos la selección de la BMU. Para ello, del conjunto de datos de entrenamiento (sección azul) se selecciona una muestra de forma aleatoria (círculo blanco) y se toma como mejor neurona del mapa (cada neurona es una intersección de la malla de líneas y, la mejor, es resaltada con un círculo amarillo intenso) la neurona que está más cerca de la muestra.

En la segunda parte, observamos el proceso de actualización de los pesos de las neuronas. En este paso, la BMU más cercana se mueve hacia la posición de la muestra, y, las neuronas dentro de su vecindario (círculo amarillo menos intenso), se acercan también a la muestra pero en un menor grado.

Por último, vemos cómo, tras un número de iteraciones, el mapa es capaz de ofrecer una aproximación de la distribución de los datos.

2.1.2. Usos del mapa auto-organizado.

El modelo del mapa auto-organizado 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 compo-

nentes son algunos de los modelos utilizados para visualizar el mapa auto-organizado.

- **Clasificación.** Una vez terminado el proceso de entrenamiento, pueden asignarse etiquetas a cada uno de los nodos y resolver problemas de clasificación dependiendo de qué BMU se active.

2.1.3. Mapa auto-organizado batch.

El proceso de entrenamiento previamente mencionado se corresponde al del mapa auto-organizado 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 [5], basándose en las propiedades matemáticas del mapa auto-organizado *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 auto-organizado *batch***.

En esta versión, la regla para la actualización de los vectores de referencia se procesan múltiples muestras del conjunto de datos de entrada, ya sea el conjunto de datos entero en cada iteración, como ocurre en nuestra implementación, o tomando un subconjunto aleatorio de muestras con reemplazo. Durante cada iteración, se recorren todas las muestras a ser evaluadas y, para cada una de ellas, se determina la BMU de forma similar a la versión *online*. A continuación, procedemos a actualizar el conjunto de vectores de referencia de la siguiente manera:

1 - Para cada muestra, X_j consideramos un subconjunto de los vectores de referencia del vector $\vec{W}$ alrededor de la BMU, al que denominamos $\vec{W_A}(i) = \{w_{BMU(i)-\sigma^2}, ..., w_{BMU(i)}, ..., w_{BMU(i)+\sigma^2}\}$

2 - Cada vector de referencia posee dos variables que nos permitirán obtener el vector de referencia para la próxima iteración mediante la realización del cociente entre ambas. Ambas serán inicializadas a 0 y cuando el vector de referencia en cuestión se encuentre dentro del rango $\vec{W_A}(i)$ para una muestra específica x, sumaremos al numerador el producto de la función de vecindario para la muestra X, su BMU y la posición que ocupa (en índices) el vector de referencia en el vector $\vec{w}$ y, en el denominador, sumaremos tan sólo el valor escalar de la función de vecindario.

$$\delta_i'(x,t) = e^{-\frac{||BMU(x)-i||}{2\sigma^2(t)}}$$

$$W_i^*(t) = \frac{\Sigma_{x \in X, BMU(x) \in \vec{W_A(i)}} \delta_i'(x,t) \cdot x}{\Sigma_{x \in X, BMU(x) \in \vec{W_A(i)}} \delta_i'(x,t)}$$

$$W_i(t+1) = \begin{cases} W_i^*(t) & si \ \ \Sigma_{x \in X, BMU(x) \in \vec{W_A(i)}} \delta_i'(x,t) > 0 \\ W_i(t) & en \ caso \ contrario. \end{cases}$$

3 - El cociente entre ambas conforma el valor del vector de referencia para la próxima iteración. Por tanto, hemos calculado nuestro nuevo vector de referencia como un promedio de las muestras de las que ha sido BMU y las muestras que han sido activadas en posiciones (en índices) del vector $\vec{W}$ cercanas ponderadas en función de esa distancia, dando más relevancia a las muestra más cercanas.

El uso del modelo *batch* frente al modelo tradicional nos va a permitir paralelizar el algoritmo de forma mucho más eficiente, ya que, mientras que, en la versión *online*, debíamos actualizar los vectores de referencia tras cada iteración, en la versión *batch* podemos evaluar un conjunto de muestras de forma simultánea y, al estar evaluando múltiples muestras y por el nuevo método de actualización de los vectores de referencia converge en un número de iteraciones considerablemente inferior al necesario en el primero. Sin embargo, como comentan *Jean-Claude Fort, Patrick Letremy y Marie Cottrell* [6], el método de entrenamiento utilizado en la versión *batch* del algoritmo presenta una mayor dependencia de la inicialización de los vectores de pesos en el instante inicial, pudiendo proporcionar clusters muy desbalanceados o una peor representación de las características topográficas que la versión *online*.

En el resto del documento nos referiremos a cada vector de referencia como pesos asociados a las neuronas y denominaremos estructura o "matriz" de pesos al vector $\vec{W}$

2.1.4. Medidas de calidad.

Para medir la calidad de un mapa auto-organizado una vez entrenado podemos utilizar dos medidas sobre el conjunto de muestras usado:

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.

$$\epsilon_q = \frac{1}{|X|} \sum_{x \in X} ||x - w_{BMU(x)}||$$

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:

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

$$\epsilon_t = \frac{1}{|X|} \sum_{x \in \vec{X}} u(x)$$

2.2. Árboles de decisión.

Un árbol de decisión [4] 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, está formado por una estructura jerárquica de reglas que nos indican a qué categoría pertenece una muestra del conjunto de datos de entrada. El 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 procede a evaluar 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 una muestra si, al ser evaluada, 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 una 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 cuál es el mejor de ellos. Este proceso es repetido hasta que se completa el árbol, es decir, todas las muestras han sido clasificadas, o, alguna de las condiciones de finalización temprana se cumpla, si es que la hubiere.

Un **punto de corte** es una combinación de un atributo del problema y un valor para el mismo con el que se va a particionar el conjunto de muestras siguiendo una relación de orden, quedando una partición para las muestras cuyo valor para el atributo sea inferior o igual al valor proporcionado y otra partición para las muestras con valores superiores (el caso de ser igual podría ser cambiado de la partición inferior a la superior siempre que se mantenga el criterio durante todo el proceso de entrenamiento y evaluación).

En las aproximaciones para entrenamiento de árboles de decisión en las que la evaluación de los puntos de corte se realiza de forma exhaustiva, es decir, analizando todos los posibles puntos de corte, y se trabaja sólo con atributos numéricos, los puntos de corte se calculan de la siguiente manera: para cada atributo y sus correspondientes valores proporcionados por las muestras, se ordenan los valores de forma ascendente o descendente y, si al realizar un recorrido secuencial sobre los valores el valor actual y su sucesor son diferentes, el punto medio entre ambos valores constituirá un punto de corte.

La calidad de cada uno de los puntos de corte obtenidos 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 partición actualmente considerada, s es el punto de corte y D_{izq} y D_{der} son las subparticiones obtenidas a partir del punto de corte.

Una medida de **impureza** [7] 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 2.1 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 2.1: Algunas medidas de impureza.

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 árbol de decisión, en el que se intentan obtener los mejores resultados, es conocer cuál ha de ser el tamaño apropiado del mismo, ya que este factor va a influir considerablemente en la calidad de las predicciones proporcionadas. Un árbol muy pequeño corre el riesgo de haber generalizado más información de la cuenta, mientras que, un árbol muy grande, puede estar demasiado especializado, dejándose influir por ruido presente en las muestras y, como consecuencia, llegar a producir la situación denominada como sobreajuste, en la que, el árbol no ha sido capaz de extrapolar los datos relevantes para la clasificación y, por ello, podría colocar muestras en clases incorrectas.

Para evitar este tipo de problemas, es común recurrir a técnicas 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, en el que 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 niveles de profundidad anteriores al nivel actualmente podado 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, que viene dado por el número de muestras cuya clase de salida original no se corresponde a la que el árbol que la evalúa predice.
  $poda(T, t)$ es el árbol obtenido de podar el subárbol t del árbol T.

  En cada iteración, se elimina el subárbol que minimiza la diferencia entre el error obtenido de la operación aplicada al subárbol t y el error al aplicarlo al conjunto total de datos, normalizado respecto de la diferencia entre el número de hojas del árbol antes y después de la poda realizada, respectivamente:

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

  Una vez generados todos los árboles T_0 a T_r se selecciona aquél 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. Para evaluar la precisión del modelo, se utiliza un conjunto reducido de datos, al que normalmente se denomina conjunto de test, que no ha sido utilizado durante el proceso de entrenamiento y del que conocemos su clase de salida de antemano.

2.2.4. Random forest.

Como explicaremos más adelante, para combinar nuestra implementación del árbol decisión en *CUDA* y el framework de computación en clúster *Spark*, optamos por realizar una implementación de un *random forest* en vez de un único árbol de decisión.

Un ***random forest*** es un conjunto de árboles de decisión, en el que el conjunto de muestras de entrenamiento es distribuido aleatoriamente en un número de árboles determinado por el usuario y, en el que, cada árbol del *random forest*, es entrenado conforme a un árbol de decisión normal. Sin embargo, la evaluación de la muestra, con el fin de obtener la clase a la que pertenece, se realiza evaluando la muestra en cada uno de los árboles que componen el *random forest* y tomando, como salida, la clase que un mayor número de árboles han predicho.

De esta manera, tenemos un modelo que, de forma similar al árbol de decisión, puede resolver problemas de clasificación o regresión no lineales, no requiere un gran preprocesamiento de las muestras antes de ser entrado para dar buenos resultados y, además, ayuda a evitar el sobreajuste, ya que, al distribuir aleatoriamente las muestras y tomar un promedio de los resultados,

ayuda a impedir que el proceso de entrenamiento se especialice en exceso hacia muestra ruidosas. Sin embargo, como ocurría en el árbol, encontrar un *random forest* óptimo global sigue siendo un problema NP-completo y, el modelo obtenido no es tan fácil de interpretar como el conjunto de reglas jerárquicas que presentaba el árbol de decisión.

Capítulo 3

Tecnologías utilizadas

3.1. 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 compañía. 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 **dispositivo** a la GPU o tarjeta gráfica utilizada.

La intercomunicación entre *host* y dispositivo sigue un modelo maestro-esclavo. El *host* actúa como maestro y es el encargado de indicar al dispositivo el código que ha de ejecutar y mandar el trabajo al 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:

- La GPU tiene muchos más núcleos *(cores)* que una CPU, lo que nos permite realizar muchas 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 disfrutar de la cantidad masiva de núcleos que tiene una GPU es necesario que ésta opere a una frecuencia más baja.
- La GPU tiene su propia estructura de memoria, que ha de usar para poder realizar operaciones. Dentro de la jerarquía de memoria encon-

tramos 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 es habitualmente realizado en *CUDA* mediante el manejo de la "memoria compartida", fundamental para obtener mejores resultados, especialmente, aquellos en los que el cuello de botella es la transferencia de datos.

- 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 y la denominada "memoria compartida" se asigna a cada bloque del grid de la GPU.

3.1.1. Python, NumPy, 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 ***NumPy***, 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.

NumPy [8] es un paquete de código abierto para Python diseñado para la computación científica. El paquete proporciona una potente estructura de datos para trabajar con arrays N-dimensionales y herramientas para realizar una gran cantidad de operaciones sobre los mismos (operaciones de cálculo matricial, algoritmos de álgebra lineal y generación de números pseudoaleatorios, entre otros).

Numba [9] es un paquete para Python, cuyo objetivo es la aceleración compilando fragmentos de código con el compilador LLVM y proporcionar la oportunidad de paralelizar código, tanto para la CPU como para la GPU. En concreto, para los dispositivos CUDA, este paquete 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 las transferencias

de memoria entre *host* y *dispositivo* o la necesidad de indicar todos los tipos a la hora de inicializar un *kernel*, entre otras ventajas.

CuPy [10] es otro paquete de Python que, por un lado y, de manera similar a Numba, nos permite generar kernels 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. 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 (CuPy y Numba) según nos interese.

Por otro lado, CuPy proporciona un API similar a la de NumPy, pero las operaciones están implementadas utilizando CUDA y, tienen algunas diferencias notables entre sí, como que: el generador de pseudoaleatorios de CuPy sólo puede usar un valor escalar y, el de NumPy, puede usar también un array con múltiples valores para generar más entropía; utilizar índices por encima del tamaño de un array genera un error en NumPy pero, en CuPy, vuelve a recorrer el array desde el principio; o que, mientras que en NumPy se puede utilizar objetos de Python, en CuPy sólo se pueden utilizar tipos de datos numéricos y booleanos.

3.1.2. Estructura de hebras, bloques y mallas.

El ***kernel*** es un fragmento de código especial, destinado a ser ejecutado en el dispositivo, en el que se indica las instrucciones qué ha de ejecutar una hebra.

```
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)
  # Optamos por 128 hebras por bloque
  tpb = 128
```

```
# Calculamos el número de bloques necesario
bloques = ejemplo.size // tpb + 1
# Lanzamos el kernel con bloques de 128 hebras cada uno
aumentar_en_1[bloques, tpb](ejemplo)
```

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

Las **hebras** son la unidad mínima en la arquitectura *CUDA*. Cada hebra es ejecutada por un núcleo *CUDA* y es consciente, en tiempo de ejecución, de su identificador dentro del bloque así como del identificador del bloque en el que se encuentra y del tamaño 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 pueden cooperar entre sí. Al conjunto de esos bloques, se le denomina **"grid" o malla**.

Tanto las hebras dentro de un bloque, como los bloques dentro de una malla, pueden tener estructuras unidimensionales, bidimensionales y tridimensionales. Las dimensiones de estas estructuras será indicada por el *host* a la hora de ejecutar el *kernel*.

Variable	**Significado**
cuda.threadIdx.[x|y|z]	Índice de la hebra dentro del bloque
cuda.blockDim.[x|y|z]	Número de hebras en el bloque.
cuda.blockIdx.[x|y|z]	Índice del bloque dentro del grid.
cuda.gridDim.[x|y|z]	Número de bloques en el grid.
cuda.grid()	Identificador único de la hebra en el grid.
cuda.gridsize()	Número total de hebras que usa el grid.

Cuadro 3.1: Variables para indexación de hebras con Numba CUDA.

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.

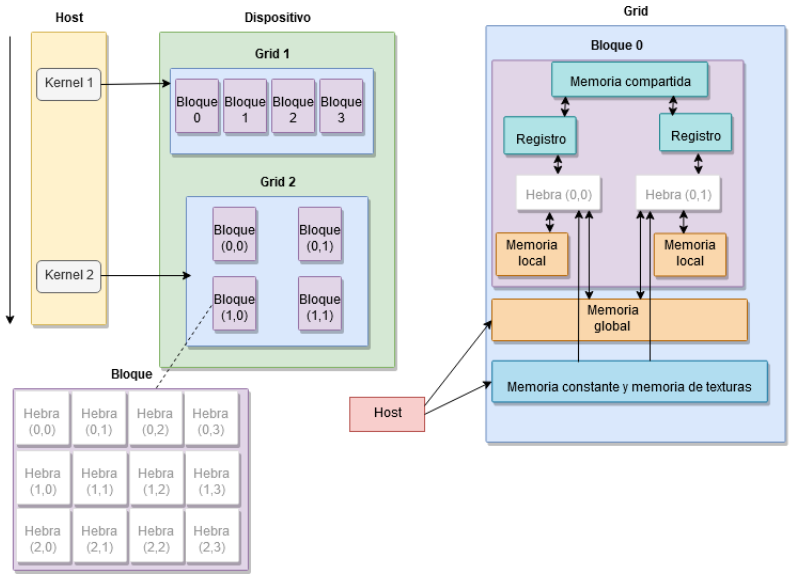

Figura 3.1: Jerarquía de hebras y memoria en CUDA.

Además, en los casos unidimensionales, el número de hebras por bloque está limitado a un máximo que depende de la tarjeta gráfica en cuestión, habitualmente 1024 hebras. No obstante, ni el uso de un único bloque de N hebras ni N bloques de 1 hebra es recomendable, ya que estaríamos dejando gran parte de la capacidad de computación paralela del dispositivo *CUDA* sin ser usada. En el primer caso, al haber un único bloque, sólo estaríamos haciendo uso de uno de los múltiples SMs que nos ofrece el dispositivo CUDA; en el segundo caso, al tener cada bloque una hebra, dado que los hebras de un bloque se ejecutan en *warps*, en cada bloque sólo se estaría utilizando 1 de las 32 hebras disponibles.

3.1.3. Estructura de memoria y 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 colindantes en memoria son colocados a través de varios niveles de caché, que tiene un tamaño más limitado que la memoria DRAM, pero con un acceso de lectura y escritura mucho más rápido.

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 3.2: Resumen de los tipos de memoria en CUDA.

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 3.2, podemos ver un resumen de los tipos de memoria existentes, dónde se pueden usar y dónde se encuentran dichos datos en el dispositivo.

Si utilizamos los arrays N-dimensionales de NumPy, no tenemos la necesidad de realizar las transferencias de memoria. No obstante, es recomendable manejarlas manualmente para evitar cualquier transferencia innecesaria de datos entre *host* y dispositivo.

3.1.4. Sincronización y operaciones atómicas.

Es frecuente la necesidad de que múltiples hebras cooperen, usando datos en alguna región de memoria del dispositivo, a la que tienen acceso de forma simultánea. En estos casos, podríamos encontrarnos ante el riesgo de una dependencia de datos de tipo *RAW (Read After Write)*, es decir, una situación en la que se lee un dato antes que los cálculos previos que necesitamos se hayan realizado. Para evitar este tipo de dependencias, *CUDA* proporciona varios mecanismos para sincronizar las hebras utilizadas, de los que vamos a destacar los dos usados en este trabajo.

Por un lado, si no estamos utilizando los *streams* de *CUDA*, característica que permite lanzar *kernels* distintos de forma concurrente, tenemos garantizado que un *kernel* no será ejecutado hasta que el *kernel* anterior no haya

Función	Definición
cuda.device_array(dimensiones, tipo)	Declara un array con las dimensiones y tipo de datos dados en memoria global. Invocada desde *host*.
cuda.to_device(array)	Envía array de la memoria del *host* a la memoria global del dispositivo. Invocada desde *host*.
nombre_array.copy_to_host()	Método para enviar nombre_array de la memoria del dispositivo a la del *host*. Invocada desde *host*.
cuda.local.array(dimensiones, tipo)	Declara un array con las dimensiones y tipo de datos dados en memoria local. Invocada desde *kernel*.
cuda.shared.array(dimensiones, tipo)	Declara un array con las dimensiones y tipo de datos dados en memoria compartida. Invocada desde *kernel*.

Cuadro 3.3: Algunas funciones para trabajar con la memoria del dispositivo en CUDA.

terminado de procesarse. El uso de múltiples *kernels* no es recomendable si no es necesario, ya que, cuando queden pocas operaciones por realizar en uno de los *kernels*, parte del dispositivo podría no estar realizando cálculo alguno, y, en el lanzamiento de cada *kernel*, existe un pequeño intervalo de tiempo desde que el *host* invoca el *kernel* y éste empieza a ser ejecutado en la GPU. No obstante, a veces, podríamos encontrar situaciones en la que la concurrencia de los *kernels* ayude a que el SM no esté ocioso, resultado de la ejecución de operaciones bloqueantes de un sólo kernel o alcanzar bifurcaciones del código donde se podrían intercalar la ejecución de 2 *streams*, que se explicarán posteriormente.

Por otro lado, dentro de un bloque, podemos sincronizar todas las hebras del mismo mediante el uso de la función ***cuda.syncthreads()***. Esta función, que es invocada desde un *kernel*, garantiza que todas las instrucciones hasta el punto de invocación han sido ejecutadas por todas las hebras del bloque. Para ello, las hebras que ya han realizado los cálculos se quedan esperando a que las otras terminen, por lo que sólo debe usarse cuando sea necesario.

Otra forma de evitar los riesgos *RAW* es el uso de operaciones atómicas. Las operaciones atómicas son instrucciones que realizan la lectura, modificación

y escritura de una posición de memoria global o compartida a la vez, es decir, está garantizado que realice todas sus operaciones antes de que otra hebra trabaje sobre la misma posición de memoria. Imaginemos el caso en el que una posición de memoria tiene un valor, 0, y dos hebras, A y B, quieren sumar dos valores: 1 y 2, respectivamente. El resultado final que deberíamos obtener sería 3, sin embargo, si no usamos las operaciones atómicas podría ocurrir que: en primer lugar, las hebras A y B leen el valor 0; en segundo lugar, A suma 1 y lo escribe en la posición de memoria; por último, B suma 2 a lo que había leído (0), con lo que en la posición de memoria queda como resultado final 2. El uso de las operaciones atómicas asegura que estas situaciones no ocurran a cambio de que la operación sea más lenta que una suma tradicional, especialmente cuando muchas hebras quieren modificar la misma posición de memoria. La suma atómica es utilizada en Numba con la función ***cuda.atomic.add(my_array, posición, valor_a_sumar)***.

3.1.5. Generación de números pseudoaleatorios en la GPU.

Numba nos proporciona un generador de números pseudoaleatorios para *CUDA*, utilizando el algoritmo *xoroshiro128+* [11], para generar números de una distribución uniforme y, el método de Box-Muller, para transformar la distribución uniforme a una distribución normal.

En la GPU, para que el generador pueda ser inicializado con una semilla, ha de generarse un estado aleatorio para cada hebra, ya que, si todas las hebras usaran el mismo, el orden en el que se ejecuten las hebras afectaría al resultado. Por ello, Numba nos proporciona la función **create_xoroshiro128p_states(n, seed)**, que se invoca desde el *host* devuelve un array en la memoria global del dispositivo con los estados aleatorios para n hebras basados en la semilla *seed*.

La función **xoroshiro128p_[distribución]_[tipo](estados, id_hebra)**, invocada desde el *kernel*, nos permite obtener números pseudoaleatorios de la distribución y tipo proporcionados. La distribución puede ser: *uniform*, para la distribución uniforme; y *normal*, para una distribución normal. Los dos tipos de datos soportados son valores en coma flotante de 32 bits *(float32)* o valores en coma flotante de 64 bits *(float64)*.

```
from numba.cuda.random import create_xoroshiro128p_states
from numba.cuda.random import xoroshiro128p_uniform_float32
import numpy as np
@cuda.jit
def pseudoaleatorios(rng_states, array):
    """
```

```
    :param rng_states Estados aleatorios.
    :param array Array a inicializar
    """
    # La hebra coge su identificador unidimensional único.
    idx = cuda.grid(1)

    # Sacamos el float32 aleatorio correspondiente.
    if idx < array.size:
        array[idx] = xoroshiro128p_uniform_float32(rng_states,
                                                   idx)

# Tamaño del array
n = 10000
# Generamos un array de flot32 sin inicializar.
mi_array = np.empty(n, dtype=np.float32)
# Generamos los estados aleatorio de todas las hebras para la semilla 7.
rng_states = create_xoroshiro128p_states(n, seed=7)
# Número de hebras por bloque
tpb = 512
# Invocamos el kernel para inicializar mi_array con pseudoaleatorios.
pseudoaleatorios[mi_array.size // tpb + 1, tpb](rng_states, mi_array)
```

Código Fuente 3.2: Inicialización pseudoaleatoria de un array.

El resultado del código 3.2, por consiguiente, parte de una serie de números aleatorios *(float32)* que cada hebra guarda en la posición de "mi_array" que coincide con su identificador de hebra, repartidas en 20 bloques de 512 hebras. ($mi\_array.size\ div\ 512 + 1 = 20$)

3.1.6. Streams.

Un **stream** en *CUDA* es una secuencia de órdenes (lanzamiento de *kernels*, transferencias de memoria, etc.) enviadas desde el *host* al dispositivo. Mientras que la secuencia de órdenes dentro de un *stream* ha de ser ejecutada secuencialmente, las órdenes entre los *streams* pueden intercalarse, potencialmente permitiendo un mejor uso de los recursos al ejecutar múltiples *kernels* de forma concurrente.

Numba nos permite utilizar esta característica con la función **cuda.stream()**, que genera un nuevo stream y nos devuelve el identificador del mismo. Para usarlo, al lanzar el kernel o hacer la transferencia de memoria hemos de pasar la variable con el identificador en el parámetro posicional "stream'. Posteriormente, podemos sincronizar todos los *streams* utilizando la función **cuda.synchronize()**.

```
...
# Generamos dos streams
stream1 = cuda.stream()
stream2 = cuda.stream()

# Declaramos dos arrays de 100000 elementos
n = 100000
array1 = np.ones(n, dtype=np.float32)
array2 = np.empty(n, dtype=np.float32)

# Hacemos las transferencias de memoria
d_array1 = cuda.to_device(array1, stream=stream1)
d_array2 = cuda.to_device(array2, stream=stream2)

tpb = 128
bloques = n // tpb + 1

# Lanzamos los kernels de forma asíncrona usando
# los straems
aumentar_en_1[bloques, tpb, stream1](d_array1)
rng_states = create_xoroshiro128p_states(n, seed=7)
pseudoaleatorios[bloques, tpb, stream2](rng_states, d_array2)

# Traemos de vuelta los arrays al host
array1 = d_array1.copy_to_host(stream=stream1)
array2 = d_array2.copy_to_host(stream=stream2)

# Sincronizamos para asegurar que ambos streams han finalizado.
cuda.synchronize()
```

Código Fuente 3.3: Uso de streams en CUDA para ejecución concurrente.

En el código fuente 3.3 obtenemos dos arrays tras la finalización del mismo. El primero de ellos, "array1" es inicializado con unos, transferido a la memoria global del dispositivo y, haciendo uso del *kernel* presentado previamente, aumentamos en 1 el contenido de los mismos, obteniendo un array de 100000 doses con el uso de 782 bloques ($d\_array1.size\ div\ 128 + 1 = 782$) de 128 hebras. Tras realizar los cálculos, los resultados son devueltos a la memoria del host.

En el caso de "array2", hemos tomado un array de 100000 elementos sin inicializar, hemos transferido el array a la memoria global del dispositivo y, haciendo uso del *kernel* para la generación de pseudoaleatorios, hemos obtenido un array cuyos elementos han sido inicializados pseudoaleatoriamente utilizando el mismo número de bloques y hebras por bloques que en el caso de "array1". Durante la ejecución, el uso de los *streams*, abre la posibilidad de que las operaciones relacionadas con "array1" y las operaciones relacionadas con "array2" puedan ser ejecutadas de forma concurrente.

3.2. 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; y proporciona herramientas de alto nivel como ML/MLib, una librería con algoritmos de *machine learning*.

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. El hecho de que el RDD sea inmutable, es decir, de sólo lectura, facilita la distribución del trabajo a lo largo del clúster aunque implica que las acciones que transforman los datos necesiten crear un nuevo *RDD* para los resultados. Existen dos formas básicas de generar un *RDD* en *Spark*: paralelizando una colección existente en el código que controla la ejecución de *Spark* o leyendo los datos de un sistema de almacenamiento.

Al poder utilizar la *API* desde *Python* podemos combinar de una manera simple el uso de *Spark* y *CUDA*. El uso de ambas nos proporciona, entre otras, las siguientes ventajas:

- *Spark* hace disponibles herramientas para la lectura de datos desde un sistema de almacenamiento.
- El código que realicemos combinando *Spark* y *CUDA* podría ser utilizado tanto en una única máquina como en un clúster.
- Si trabajamos con una única máquina, podemos solucionar problemas de escalabilidad ajustando el número de particiones (subdivisiones del RDD) y el número de ejecutores (número de procesos que podrían ejecutarse simultáneamente) para, por ejemplo, evitar quedarnos sin memoria *RAM* en el dispositivo.

La distribución de trabajo en Spark en este proyecto se realizará utilizando la transformación *mapPartitions* del *RDD*, que generará un nuevo RDD a partir de los resultados obtenidos al aplicar la función pasada a *mapPartitions* como parámetro.

```
import numpy as np
from pyspark.sql import SparkSession

# Cogemos el contexto de Spark
```

```
spark = SparkSession.builder.master("local").appName("ejemplo").getOrCreate()
sc = spark.sparkContext

# Generamos un array de 100000 ceros.
mi_array = np.zeros(100000, np.float32)

# Creamos un RDD a partir del array
rdd = sc.parallelize(mi_array)

# Número de hebras por bloque
tpb = 128

# Encapsulamos un kernel en una función de Python
def gpu_work(data):
  # Transformamos los datos a un array de NumPy
  gpu_data = np.asarray(list(data), dtype=np.float32)
  # Invocamos el kernel utilizado en los ejemplos previos.
  aumentar_en_1[gpu_data.size // tpb + 1, tpb](gpu_data)
  # Devolvemos los datos calculados
  return gpu_data


# Realizamos la transformación del RDD
output = rdd.mapPartitions(gpu_work)

# Recogemos los resultados en un array de NumPy
output_np = output.collect()
```

Código Fuente 3.4: Uso de Spark con Numba.

En el código 3.4 inicializamos un array de 100000 ceros *(*float32) usando *NumPy*. Mediante el uso de la función **parallelize** de *PySpark*, generamos un RDD a partir de la colección. La función *gpu_work* va a ser ejecutada en cada partición del RDD, potencialmente almacenadas en diferentes nodos del clúster, que lanzará el mínimo número de bloques con 128 hebras por bloque para invocar el *kernel aumentar_en_1*. Utilizando **mapPartitions** y **collect** de *PySpark*, en primer lugar indicamos la función a ejecutar sobre cada partición del RDD y con la segunda iniciamos el procesamiento de la misma y almacenamos los resultados. Por tanto, *output_np*, será un array que contendrá 100000 unos.

Capítulo 4

Estado del arte: trabajos relacionados.

La paralelización en *CUDA* de los mapas auto-organizados de Kohonen y de los árboles de decisión son problemas que han sido previamente estudiados en la literatura.

En ***Parallel High Dimensional Self Organizing Maps Using CUDA***, Codevilla, Bothelo, Filho y Gaya [12] proponen una implementación en *CUDA* para la formulación tradicional del mapa auto-organizado de Kohonen. En ella, proponen una versión en la que cada iteración se subdivide 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, es decir, un reparto de los cálculos necesarios para obtener la distancia euclídea que depende de un parámetro indicado por el usuario; una reducción, para encontrar la mejor distancia entre las neuronas; y un método para la adaptación de pesos de neuronas cuyo grado de paralelismo depende de dimensión del problema (tamaño de una neurona).

En ***Parallel Batch Self-Organizing Map on Graphics Processing Unit Using CUDA***, Daneshpajouh, Delisle, Boisson, Krajecki y Zakaria [13] plantean una adaptación en CUDA para la versión iterativa de cómputo en *batchs* del mapa auto-organizado de Kohonen. En ella, aprovechan las capacidades de concurrencia disponibles en los dispositivos CUDA, paralelizando parte del algoritmo y dejando otra parte para ser realizada con la CPU.

Con respecto a los árboles de decisión, ***CUDT: a CUDA based decision***

tree algorithm, de Lo, Chang, Sheu, Chiu y Yuan [14], 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 árboles de decisión cuyo objetivo es la clasificación de problemas con respuesta binaria.

Aparte de la aproximación por especialización presentada en el trabajo anterior, existe otra alternativa, más frecuente, versátil y aplicable a diferentes tipos de problemas a resolver, se basa en 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 [15], donde el objeto principal de su trabajo es paralelizar en *CUDA* los cálculos asociados a los histogramas utilizados en ***Streaming Parallel Decision Trees***, de Ben-Haim y Tom-Tov [16], un algoritmo para la paralelización de árboles para CPU.

Capítulo 5

Implementación.

5.1. 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, *Nsight*, sobre un ejemplo razonable para evaluar el rendimiento del algoritmo.

5.2. Desarrollo del mapa auto-organizado de Kohonen.

Para implementar los mapas auto-organizados de Kohonen, primero consideramos la versión tradicional *online* y, posteriormente, tras ver las limitaciones de la primera, evaluamos la versión computada en *batchs*, que ha sido implementada tanto para CPU, usando *NumPy*, como para CUDA, usando *Numba*.

5.2.1. Limitaciones del mapa auto-organizado online.

Mientras que la implementación del mapa auto-organizado *online* fue el punto de partida para la realización de este trabajo, tuvimos que descartar esta

versión del algoritmo, ya que, el objetivo de este trabajo es resolver problemas con un gran número de muestras utilizando *CUDA* y *Spark*.

En esta versión, en cada iteración, se selecciona una única muestra del conjunto de datos y ésta es evaluada para actualizar los pesos de las neuronas, que serán el punto de partida de la siguiente iteración, limitando a una el número de muestras que pueden procesarse a la vez y, por tanto, secuencializando el proceso.

La opción más apropiada para paralelizar esta versión del algoritmo sería procesar una única muestra usando tantas hebras como neuronas tenga el mapa de salida. En ese caso, en cada iteración, cada hebra podría calcular su distancia euclídea de la muestra con los pesos de la neurona asociada a la hebra, usaríamos el algoritmo de la reducción, que explicaremos posteriormente, para encontrar la BMU y, cada hebra, realizaría la actualización de los pesos de su neurona, si ese fuera el caso. Sin embargo, este procedimiento sólo conseguiría ganancias significativas con respecto a su versión para CPU con un mapa de neuronas considerablemente grande, factor que no parece razonable en un algoritmo cuyo principal uso es aprovechar las ventajas que proporciona el *clustering* para procesar cantidades muy grandes de datos.

Determinadas estas limitaciones y, dado que nuestro objetivo es evaluar un conjunto con un número de muestras elevado, optamos por implementar la versión del mapa auto-organizado que nos permite evaluar múltiples muestras simultáneamente, conocido como el mapa auto-organizado *batch*.

Para el desarrollo de esta versión del algoritmo hemos combinado el uso de *CUDA* mediante *Numba* y *Spark*. En primer lugar, vamos a ver un esquema general del uso de *Spark* para afrontar nuestro algoritmo iterativo y, a continuación, explicaremos en detalle la implementación de los *kernels* para *CUDA*.

5.2.2. Uso de Spark.

Utilizar *Spark* para implementar este algoritmo nos permite afrontar problemas de tamaños superiores a la capacidad de memoria de nuestro dispositivo, siempre que la memoria necesaria para evaluar una partición del *RDD* quepa en la memoria del dispositivo, y llevar la implementación realizada a un clúster con múltiples nodos, si cada nodo tiene acceso a un dispositivo *CUDA* con todas las dependencias de *software* instaladas.

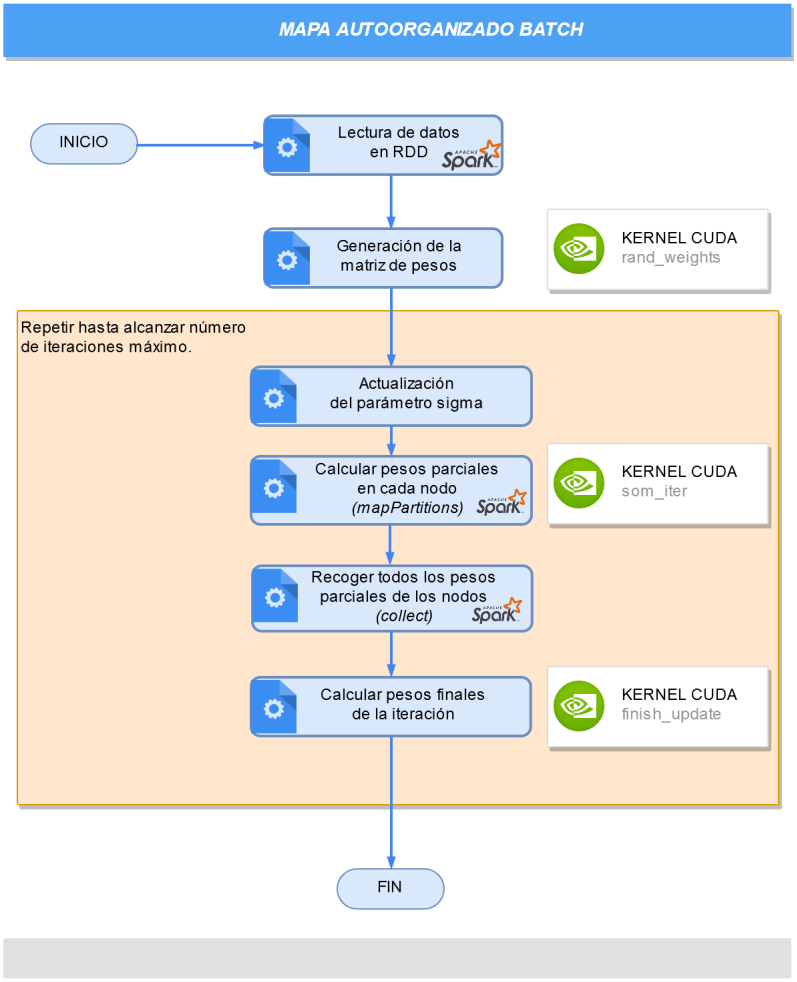

Figura 5.1: Diagrama de flujo del mapa auto-organizado desarrollado.

El algoritmo comienza con un único nodo de *Spark*, utilizando el primer kernel desarrollado, *rand_weights*, para de esta forma inicializar de manera pseudoaleatoria los valores de la estructura de pesos, en función de una semilla proporcionada por el usuario. Con esta estructura ya generada, empieza el proceso iterativo en el que:

1. Calculamos el parámetro de control σ para la iteración en función de las ecuaciones correspondientes.

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

2. Utilizando la transformación *mapPartitions*, en cada partición del *RDD*

se aplica la función *gpu_work_iter*, que encapsula el segundo *kernel* desarrollado, *som_iter*. Este *kernel* se encarga de evaluar los pesos parciales para cada neurona en función de las muestras asociadas a la partición del *RDD*. Puesto que la actualización de pesos es una división entre una sumatoria de vectores, con cada vector del tamaño de una muestra, y una sumatoria de números reales, el objetivo de cada partición será calcular esos numeradores y denominadores, a los que nos referiremos de ahora en adelante como numeradores y denominadores parciales.

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

3. Para finalizar la iteración, *Spark* reúne mediante *collect* las numeradores y denominadores parciales obtenidos y, usando el último *kernel* implementado, *finish_update* obtiene los pesos finales de la iteración.

Este proceso, que consta de 3 fases, es realizado hasta alcanzar el número máximo de iteraciones. Hemos de destacar que todas las particiones han de partir de la mismos pesos en cada iteración para realizar los cálculos. Por ello, al inicio de la iteración, es necesario distribuir la matriz de pesos a cada nodo de *Spark* que realiza esos cálculos y, al final de la iteración, reunir todos los numeradores y denominadores parciales en un único nodo, permitiéndonos obtener los pesos finales de la iteración.

Para que *Spark* pueda realizar esa distribución a lo largo de un clúster es necesario que, al final de la iteración, se haga la transferencia desde la memoria de dispositivo al *host* ,de los pesos parciales y, al inicio de la iteración, se haga la transferencia desde el *host* al dispositivo de la pesos de las neuronas correspondientes a esa iteración.

```
def spark_gpu_batch_som(rdd_data, d, max_iters, rows, cols, smooth_iters=None,
                        sigma_0=10, sigma_f=0.1, tau=400, seed=None, tpb=128):
    """
    :param rdd_data RDD con el conjunto de muestras a evaluar.
    :param d Tamaño de una muestra, dimensión del problema.
    :param max_iters Número de iteraciones a realizar.
    :param rows Número de filas en el mapa de neuronas.
    :param cols Número de columnas en el mapa de neuronas.
    :param smooth_iters Número de iteraciones en las que el parámetro
           sigma decrece siguiendo una función gaussiana.
    :param sigma_0 Valor de sigma inicial.
    :param sigma_f Valor de sigma tras alcanzar la iteración smooth_iters.
    :param tau Valor de tau para la función gaussiana.
    :param seed Semilla pseudoaleatoria para la generación inicial de pesos.
    :param tpb Número de hebras por bloque para la inicialización de pesos y
```

```
        la actualización final de los pesos.
    """
    # 1. Declaramos la estructura de los pesos.
    d_weights = cuda.device_array((rows, cols ,d), np.float32)

    # 1.2 Usamos Numba para generar los pesos de forma pseudoaleatoria.
    rng_states = create_xoroshiro128p_states(rows * cols * d, seed=seed)
    rand_weights[(d_weights.size) // tpb + 1, tpb](rng_states, d_weights)

    # 1.3 Traemos los pesos de la memoria de la GPU a la memoria del host.
    weights = d_weights.copy_to_host()

    # 2. Inicio del proceso iterativo
    for t in range(max_iters):
        # 2.a Actualizamos sigma en función de los tau y la iteración.
        if smooth_iters is None or t < max_iters:
            sigma = sigma_0 * math.exp((-t/tau))
        else:
            sigma = sigma_f

        sigma_squared = sigma * sigma

        # 2.b Cálculos parciales con mapPartitions en cada nodo.
        out = rdd_data.mapPartitions(gpu_work_iter(weights, sigma_squared))

        # 2.c En un único nodo calculamos las sumas parciales.
        out = out.collect()
        finish_update[rows*cols//tpb + 1, tpb](weights, np.concatenate(out),
                                               len(out) // 2)

    # 3. Devolvemos los pesos obtenidos
    return weights
```

Código Fuente 5.1: Uso de Spark para entrenar el mapa auto-organizado.

5.2.3. Representación de la estructura de pesos de las neuronas.

La estructura que contiene los pesos de las neuronas, que durante la ejecución de los *kernels* se encontrará almacenada en la memoria global del dispositivo, se corresponde a un array tridimensional. El primer eje indica la fila que ocupa la neurona en el mapa, el segundo eje indica la columna que ocupa la neurona en el mapa y el último eje la característica del problema a la que queremos acceder.

Mientras que nosotros podemos hacer uso de este sistema de indexación tridimensional gracias a Numba, en realidad, en el dispositivo CUDA se trata de un array unidimensional *row-major*.

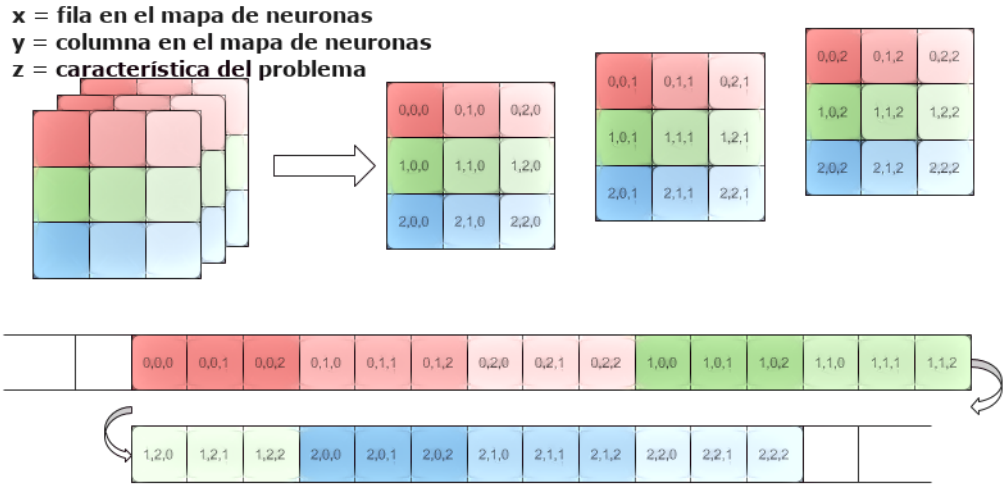

Figura 5.2: Representación de un array 3D como un array 1D row-major.

5.2.4. Kernels implementados.

5.2.4.1. Generación pseudoaleatoria de pesos de neuronas.

```python
@cuda.jit
def rand_weights(rng_states, d_weights):
    """
    Kernel para inicializar aleatoriamente la estructura de pesos con
    valores en el intervalo [0, 1) tomados de una distribución uniforme
    :param rng_states Estados aleatorios.
    :param d_weigths Vector de filas * columnas * d valores que contendrá
           los pesos asociados a las neuronas.
    """
    # La hebra coge su identificador unidimensional único.
    idx = cuda.grid(1)

    # La hebra calcula en función del su índice
    # y cocientes y restos de divisiones entereas
    n_rows, n_cols, d = d_weights.shape

    # Cálculo de la fila (eje X).
    row = idx // (n_cols * d)

    # Cálculo de la columna (eje Y).
    col_d = idx % (n_cols * d)
    col = col_d // d
    # Cálculo de la característica (eje Z).
    i = col_d % d

    # Sacamos el aleatorio correspondiente.
    if idx < d_weights.size:
        d_weights[row, col, i] = xoroshiro128p_uniform_float32(rng_states, idx)
```

Código Fuente 5.2: Inicialización pseudoaleatoria de los pesos de las neuronas.

Este proceso (código fuente 5.2) 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. Una vez generados, son trasladados de vuelta a la CPU para ser distribuidos a todos los nodos ejecutores de *Spark*.

Al lanzar este kernel, se utilizan tantas hebras como números aleatorios (tabla 5.1), distribuidos en bloques de un tamaño indicado por el usuario.

Kernel	**Bloques**	**Hebras por bloque**
rand_weights	$(filas \cdot columnas \cdot d)$ // $tpb + 1$	tpb

d = dimensión del problema.
tpb = hebras por bloque (indicados por el usuario).
// = cociente de división entera.

Cuadro 5.1: Parámetros para el lanzamiento del kernel rand_weights.

5.2.4.2. Cálculo de los numeradores y denominadores parciales.

```
def gpu_work_iter(weights, sigma_squared):
    def _gpu_work(data):
        # 1. Procesamos el dataset
        inp = np.asarray(list(data), dtype=np.float32)
        rows, cols, d = weights.shape
        nneurons = rows * cols

        # 2. Pasamos los datos a las memorias del dispositivo
        d_samples = cuda.to_device(inp)
        d_weights = cuda.to_device(weights)
        nums = np.zeros(rows * cols * d, np.float32)
        denums = np.zeros(rows * cols, np.float32)
        d_nums = cuda.to_device(nums)
        d_denums = cuda.to_device(denums)

        # 3. Tomamos el número de hebras por bloque
        if nneurons > 1024:
            raise Exception('Número de neuronas superior al límite')
        # Número de hebras necesario para que funcione la reducción.
        tpb = max(64,2**(math.ceil(math.log2(nneurons))))
        # 4. Lanzamos el kernel.
        # Memoria compartida para almacenar una muestra por bloque
        sm_size = 4 * d
        som_iter[N, tpb, 0, sm_size](d_samples, d_weights, d_nums, d_denums,
                                     sigma_squared)

        return d_nums.copy_to_host(), d_denums.copy_to_host()
    return _gpu_work
```

Código Fuente 5.3: Función a ejecutar con mapPartitions.

Una vez obtenidos los pesos iniciales de una iteración, el siguiente paso es utilizar *mapPartitions* para obtener los pesos parciales de cada partición del *RDD*, como veíamos en el código fuente 5.1. La función utilizada en cada partición se denomina *gpu_som_iter* y encapsula el lanzamiento del kernel *som_iter* y las transferencias de memoria entre host y dispositivo en cada iteración.

Kernel	Bloques	Hebras por bloque
som_iter	Nº de muestras.	$max(64, 2^{techo(\log_2 N^{o}neuronas)})$

techo=menor número entero mayor o igual que un número real.

Cuadro 5.2: Parámetros para el lanzamiento del kernel som_iter.

El *kernel som_iter* es la parte más importante de la implementación del algoritmo y realiza todas las operaciones necesarias para obtener los nume-

radores y denominadores parciales de la iteración.

```
@cuda.jit
def som_iter(d_samples, d_weights, d_nums, d_denums, sigma_squared):
    """
    :param d_samples Conjunto de todas las muestras a evaluar.
    :param d_weigths Vector de filas * columnas * d valores que contendrá
           los pesos asociados a las neuronas.
    :param d_nums Vector con los numeradores para el cálculo de la fórmula.
    :param d_denums Vector con los denominadores para el cálculo de la fórmula.
    :param sigma_squared Valor de sigma al cuadrado para el cáculo del vecindario.
    """
    nrows, ncols, d = d_weights.shape
    nneurons = nrows * ncols

    sample_idx, nueron_idx = cuda.blockIdx.x, cuda.threadIdx.x
    neuron_row, neuron_col = neuron_idx // ncols, neuron_idx % ncols
    blockSize = cuda.blockDim.x

    # 0. Declaramos la memoria compartida
    shared_sample = cuda.shared.array(shape=0, dtype=numba.float32)
    shared_distances = cuda.shared.array(shape=1024, dtype=numba.float32)
    shared_idx = cuda.shared.array(shape=1024, dtype=numba.int32)

    # 1.a Cada hebra pone una posición de la muestra en memoria compartida.
    # El bucle for permite realizar esto si la dimensión del problema fuese
    # superior al número de neuronas.
    for i in range(d // nneurons + 1):
        i_stride = i * nneurons
        my_pos = i_stride + cuda.threadIdx.x
        # Si la posición que corresponde a la hebra no supera el
        # tamaño de la muestra a cargar.
        if my_pos < d:
            shared_sample[my_pos] = d_samples[sample_idx, my_pos]
    # Sincronizamos para asegurar que la muestra ha sido cargada.
    cuda.syncthreads()

    # 1.b Calculamos las distancias euclídeas que nos corresponden.
    if neuron_idx < nneurons:
        shared_distances[neuron_idx] = 0.0
        for i in range(d):
            i_distance = shared_sample[i] - d_weights[neuron_row, neuron_col, i]
            shared_distances[neuron_idx] += i_distance * i_distance
    # Si hay más hebras que neuronas inicializamos a infinito para la reducción.
    else:
        shared_distances[neuron_idx] = np.inf

    # 1.c Inicializamos el array de índices para la reducción.
    shared_idx[neuron_idx] = neuron_idx
    # Sincronizamos para asegurar los arrays han sido inicializados.
    cuda.syncthreads()
```

Código Fuente 5.4: Primer fragmento [Cálculo de distancias] de som_iter.

El kernel comienza con la declaración e inicialización de la memoria compartida.

En primer lugar, cada hebra contribuye a cargar una característica de la muestra a evaluar por el bloque hasta la muestra ha sido cargada por completo. En segundo lugar, generamos dos arrays adicionales en memoria compartida, que serán utilizados posteriormente para calcular la BMU. Puesto que hemos limitado nuestra implementación a funcionar con un máximo de 1024 neuronas, que es el máximo de hebras por bloque, estos dos arrays serán siempre de esta dimensión. Uno de ellos, que será de *floats* de 32 bits, contendrá las distancias entre la muestra que cargamos en memoria compartida y los pesos de cada neurona del mapa. El segundo, que será de enteros de 32 *bits*, será inicializados con los índices de cada neurona. Para realizar el cálculo de la distancia euclídea, cada hebra calculará su distancia con la neurona que le corresponde y la muestra cargada en memoria compartida. Si hubiese más hebras en el bloque que neuronas en el mapa, el resto de distancias son inicializadas a infinito.

Puesto que nuestro siguiente objetivo será encontrar la BMU, es decir, la neurona con menor distancia, no es necesario calcular la raíz cuadrada de la división euclídea, ya que ésta no afecta a la relación de orden. Para encontrar la distancia mínima, utilizamos un algoritmo frecuentemente utilizando en la GPU: **la reducción**.

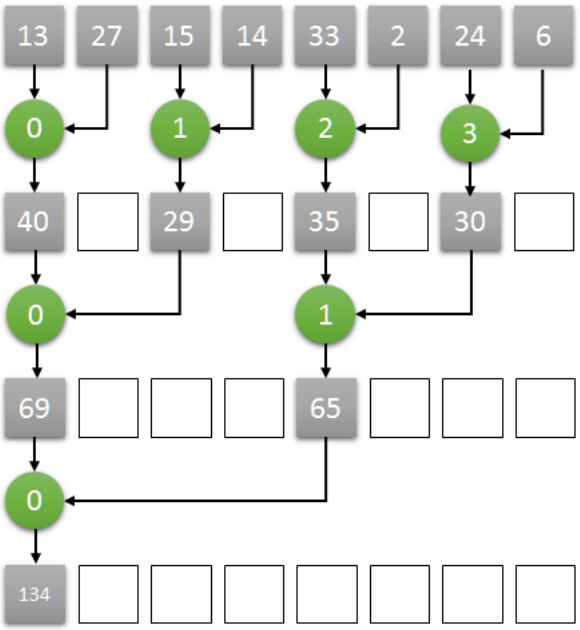

Figura 5.3: Una reducción paralela de una sumatoria.

```
    # Recorrido de árbol de hojas a la raíz (posición 0)
    if blockSize >= 1024 and neuron_idx < 512:
        if shared_distances[neuron_idx + 512] < shared_distances[neuron_idx]:
            shared_distances[neuron_idx] = shared_distances[neuron_idx + 512]
            shared_idx[neuron_idx] = shared_idx[neuron_idx + 512]
    cuda.syncthreads()

    if blockSize >= 512 and neuron_idx < 256:
        if shared_distances[neuron_idx + 256] < shared_distances[neuron_idx]:
            shared_distances[neuron_idx] = shared_distances[neuron_idx + 256]
            shared_idx[neuron_idx] = shared_idx[neuron_idx + 256]
    cuda.syncthreads()

    if blockSize >= 256 and neuron_idx < 128:
        if shared_distances[neuron_idx + 128] < shared_distances[neuron_idx]:
            shared_distances[neuron_idx] = shared_distances[neuron_idx + 128]
            shared_idx[neuron_idx] = shared_idx[neuron_idx + 128]
    cuda.syncthreads()

    if blockSize >= 128 and neuron_idx < 64:
        if shared_distances[neuron_idx + 64] < shared_distances[neuron_idx]:
            shared_distances[neuron_idx] = shared_distances[neuron_idx + 64]
            shared_idx[neuron_idx] = shared_idx[neuron_idx + 64]
    cuda.syncthreads()

    if neuron_idx < 32: # Unroll de warp. No necesitamos sincronizar.
        if shared_distances[neuron_idx + 32] < shared_distances[neuron_idx]:
            shared_distances[neuron_idx] = shared_distances[neuron_idx + 32]
            shared_idx[neuron_idx] = shared_idx[neuron_idx + 32]
        if shared_distances[neuron_idx + 16] < shared_distances[neuron_idx]:
            shared_distances[neuron_idx] = shared_distances[neuron_idx + 16]
            shared_idx[neuron_idx] = shared_idx[neuron_idx + 16]
        if shared_distances[neuron_idx + 8] < shared_distances[neuron_idx]:
            shared_distances[neuron_idx] = shared_distances[neuron_idx + 8]
            shared_idx[neuron_idx] = shared_idx[neuron_idx + 8]
        if shared_distances[neuron_idx + 4] < shared_distances[neuron_idx]:
            shared_distances[neuron_idx] = shared_distances[neuron_idx + 4]
            shared_idx[neuron_idx] = shared_idx[neuron_idx + 4]
        if shared_distances[neuron_idx + 2] < shared_distances[neuron_idx]:
            shared_distances[neuron_idx] = shared_distances[neuron_idx + 2]
            shared_idx[neuron_idx] = shared_idx[neuron_idx + 2]
        if shared_distances[neuron_idx + 1] < shared_distances[neuron_idx]:
            shared_distances[neuron_idx] = shared_distances[neuron_idx + 1]
            shared_idx[neuron_idx] = shared_idx[neuron_idx + 1]
    cuda.syncthreads()

    # La mejor neurona se encuentra en la posición 0 del array.
    bmu = shared_idx[0]
    bmu_row, bmu_col = bmu // ncols, bmu % ncols
    cuda.syncthreads()
```

Código Fuente 5.5: Segundo fragmento [Reducción] del kernel som_iter.

La reducción puede utilizarse para obtener el resultado de aplicar un operador binario a lo largo de un array, siempre que el operador en cuestión cumpla la propiedad asociativa. En nuestro caso, dicho operador es el mínimo entre dos elementos. Para realizar esta operación de manera eficiente dentro de un bloque, se simula un recorrido hacia arriba sobre un árbol binario balanceado (figura 5.3), en el que vamos aplicando la operación sobre los dos hijos y guardando el resultado en el nodo padre, tomando los distancias cargadas en memoria compartida como las hojas y alcanzado el resultado final en la raíz. Por ello, era necesario que el número de hebras por bloque fuese potencia de 2, si teníamos más hebras que neuronas completábamos las distancias con infinito, que actúa como elemento neutro de la operación mínimo, y, añadíamos un array extra con los índices para propagar la posición con el mejor índice mientras hacemos el recorrido. Podemos consultar con más detalle cómo realizar una implementación de una reducción de alto rendimiento en *CUDA* en la referencia bibliográfica [17].

Para finalizar el *kernel*, se realiza el cálculo de los numeradores y denominadores parciales. Para ello, cada hebra del bloque se corresponde con una neurona y mide la distancia euclídea que existe entre la posición de la BMU y la posición de la neurona en el mapa. Si esa distancia es menor o igual que el parámetro de control σ^2, se realiza la suma del vector del numerador con el producto de esa distancia y la muestra guardada en la memoria compartida del bloque y sólo la distancia con el denominador.

```
# 3. Realizamos la actualización de los pesos.
    if neuron_idx < nneurons:
        dist = (neuron_row - bmu_row) * (neuron_row - bmu_row) + \
               (neuron_col - bmu_col) * (neuron_col - bmu_col)
        # Si estamos dentro del rango de actualización.
        if dist <= sigma_squared:
            hck = math.exp(-dist/(2 * sigma_squared))
            # Guardamos sumatoria del denominador.
            cuda.atomic.add(d_denums, neuron_row * ncols + neuron_col, hck)
            # Guardamos sumatoria del numerador.
            for i in range(d):
                cuda.atomic.add(d_nums, neuron_row*ncols*d + neuron_col*d+i,
                                hck * shared_sample[i])
```

Código Fuente 5.6: Tercer y último fragmento del kernel som_iter.

Puesto que múltiples hebras pueden tener la misma BMU y, por tanto, estar actualizando las mismas posiciones en memoria a la vez se utilizan **operaciones atómicas**, que evitan las condiciones de carrera que pueda surgir a cambio de una mayor latencia en la operación. Hemos de indicar que, para las operaciones atómicas, necesitamos trabajar con arrays unidimensionales,

por lo que hemos de hacer los cálculos de indexación necesarios para acceder a las posiciones de memoria deseadas.

5.2.4.3. Cálculo de los pesos finales de la iteración.

Una vez todos los resultados han sido recopilados en un nodo de *Spark*, lanzamos el *kernel finish_update*, que realizará la sumatoria de los numeradores parciales y de los denominadores parciales para cada neurona así como la división entre ambos. Si ninguna muestra activó la neurona en cuestión, es decir, la suma de todos sus denominadores parciales es 0, se mantendrán los pesos de la iteración anterior para esa neurona. En caso contrario, los pesos de la neurona se corresponden con el vector obtenido de la división. Para lanzar este *kernel* se utilizan tantas hebras como neuronas hay en el mapa, divididas en bloque de *tpb* hebras. Cada hebra realiza los cálculos asociados a una neurona.

Kernel	**Bloques**	**Hebras por bloque**
finish_update	(Nº de neuronas // *tpb* + 1)	*tpb*

Cuadro 5.3: Parámetros para el lanzamiento del kernel finish_update.

```
@cuda.jit
def finish_update(d_weights, partials, numParts):
    """
    :param d_weights Array de pesos de neuronas.
    :param partials Array con sumas parciales.
    :param numParts Número de resultados parciales a procesar.
    """
    idx = cuda.grid(1)
    nrows, ncols, d = d_weights.shape
    if idx < nrows * ncols:
        row, col = 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]

        # b) Si no es 0 el denominador realizamos la división y cambiamos pesos.
        if partials[numsize + row * ncols + col] != 0:
```

```
        for k in range(d):
            d_weights[row, col, k] = partials[row*ncols*d + col*d +k] / \
                                     partials[numsize + row * ncols + col]
```

Código Fuente 5.7: Actualización final de la matriz de pesos.

5.3. Desarrollo de un modelo de árbol de decisión.

La implementación del modelo de árbol de decisión se basa en CUDT [14], que a su vez se fundamenta en SPRINT [18] y la operación de *scan*.

5.3.1. Lista de atributos.

Una lista de atributos, es una estructura auxiliar, procedente de SPRINT [18], 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 5.4: Una lista de atributos sin ordenar.

Las columnas de la tabla 5.4 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 corresponde a la etiqueta de salida asociada a la muestra de la fila.
- **ID Muestra**, que se corresponde al identificador de la muestra. Al principio, se corresponde 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". En la implementación realizada, se utiliza un array para cada columna.

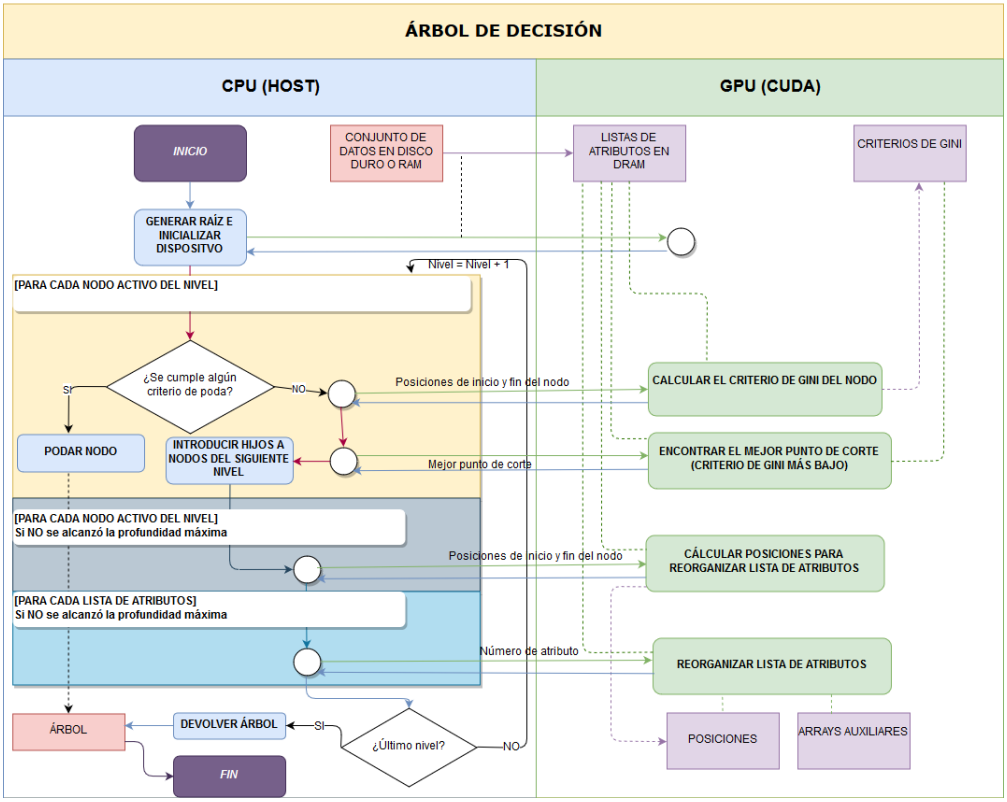

Figura 5.4: Diagrama de flujo de la implementación del árbol de decisión.

5.3.2. 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 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.

5.3.3. 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 una primitiva ampliamente utilizada en el mundo de la GPU denominada *scan*.

El *scan* [19], 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 el *scan inclusivo*. Por tanto, para los propósitos de este documento, cada vez que hablemos de *scan*, nos estaremos refiriendo al *scan inclusivo*, que al aplicarla sobre un array, nos devuelve lo siguiente:

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

La implementación realizada utilizada operaciones directas sobre *warps*, permitiendo que una ejecución más rápida al realizar todas las operaciones directamente sobre los registros del multiprocesador. De manera más específica el procedimiento realizado es el siguiente.

1. Se lanza un kernel con una estructura con un número de hebras por bloque predeterminado por el usuario y el mínimo número de bloques para procesar todo el array.

2. El objetivo de cada bloque es calcular su *scan* local.

2.1 En primer lugar, cada uno de los *warps* del bloque calcula su *scan* local y lo almacena en un array. La operación utilizada para manejar los *warps* es **shfl_up_sync**. Esta operación nos permite realizar una copia y desplazamiento hacia la derecha en el warp en función de una máscara, un valor y un desplazamiento. La máscara nos permite es un conjunto de 32 bits que nos permite indicar cuáles de las 32 hebras han de ser usadas. Para nuestra operación, la máscara está activa para todos las hebras. El valor indica qué valores vamos a utilizar para realizar la operación y ese valor será devuelto en caso de salirnos del rango del warp. Por último, el desplazamiento nos sirve para calcular cuántas posiciones nos hemos de desplazar hacia la derecha. Por ello, empezamos con un desplazamiento de 1 y vamos ampliando siguiendo las potencias binarias inferiores a 32. Para que los resultados ya calculados no sean modificados, nos aseguramos de trabajar sólo con las hebras dentro del warp con índice superior o igual al desplazamiento. Así, al tomar el desplazamiento 1, todos los elementos se suman a su anterior y el primero se mantiene constante. Al tomar el desplazamiento 2, sólo tenemos qué realizar la suma del que había dos posiciones antes, y así, sucesivamente, hasta llegar al último valor. Los últimos valores de cada *scan* local serán guardados en un array en memoria compartida.

2.2 Una vez las sumas locales de los warps han sido realizadas, hemos de añadir la suma acumulada obtenida en los *warps* previos para obtener el resultado final del bloque. Para ello, se realiza un *scan* de los *warps* previos, obteniendo el array con las sumas acumuladas previas y estas son aplicadas a los elementos del array correspondientes.

3 Una vez tenemos el *scan* de cada bloque, hemos de realizar un procedimiento similar para llevar el *scan* local del bloque a todo el array. Esto ha sido realizado, o bien mediante sumas atómicas en el mismo *kernel*, que son más lentas que las operaciones normales, o bien combinando el trabajo con otros *kernels* que tenían que ser lanzados de forma independiente.

5.3.4. 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:

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

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

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

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

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:

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

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.

5.3.5. 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 campo "ID Muestra", podemos generar fácilmente un array de booleanos donde cada elemento indica si la muestra con el ID asociado a su posición.

Si aplicamos la operación de *scan* sobre este array auxiliar, la suma acumulada en cada posición nos indica el número de elementos que pertenecen al nodo hijo codificado con la etiqueta positiva. Por lo que, partiendo de que previamente estaban ordenados podemos hacer la subdivisión y mantener el orden teniendo en cuenta que:

1. Si el elemento en cuestión pertenece al nodo hijo izquierda (codificado como positivo en el array auxiliar), su nueva posición en la lista de atributos sería la suma acumulada obtenida en el array auxiliar menos uno (para que el primer índice sea el 0).
2. En caso contrario, la nueva posición es el número total de elementos en el nodo hijo izquierda (para que ambos queden separados) más la diferencia de ese total de elementos del nodo hijo izquierda con la suma acumulada del array auxiliar (que nos indicaría cuántos elementos de la clase negativa llevamos hasta el momento) menos uno (porque la indexación empieza en 0).

5.3.6. Representación del árbol.

Durante el proceso de entrenamiento, la necesidad de lanzar múltiples *kernels* al no disponer desde Python de paralelismo dinámico ha hecho que hayamos optado por almacenar tanto la estructura del árbol de salida como la que controla los nodos a evaluar en la CPU.

La estructura que controla los nodos a evaluar es una lista que contiene todos los nodos activos del nivel y, cada elemento de la lista, es una tupla que contiene el inicio, el fin del nodo y el índice correspondiente a ese nodo si recorremos el nivel de izquierda a derecha y no se hubiese podado ningún elemento. Puesto que es necesario que la CPU acceda a estos datos para controlar el flujo de ejecución de los *kernels* no tiene sentido almacenar la misma en la GPU.

En el caso del árbol de salida, deberíamos de almacenar una estructura que nos permita almacenar los valores y atributos de un punto de corte y la clase de salida, si fuese un nodo terminal. Esta estructura podría ser almacenada

por la GPU y podría ayudar a obtener mejores resultados, especialmente si disponemos de paralelismo dinámico. Sin embargo, tanto el momento en el que se poda un nodo como en el que se calcula el mejor punto de corte suponen la finalización de un *kernel* y la devolución de datos a la CPU, por lo que añadir un dato más no supondría el cuello de botella si no la mera parada para realizar la transferencia. Además, en temas de escalabilidad, no almacenar la estructura del árbol en la GPU evita un gasto considerable de memoria RAM del dispositivo en un algoritmo con una gran complejidad espacial debido al uso de las listas de atributos en la GPU, que añaden dos campos extras para cada combinación de muestra y atributo que no sea la etiqueta de salida.

La representación utilizada para el árbol de salida es una lista de diccionarios de tuplas. En primer lugar, la lista tendrá tantos diccionarios como niveles de profundidad tenga el árbol, siendo el nivel 0 la raíz. El diccionario tendrá como clave de entrada un valor numérico, que se corresponderá a la posición que ocuparía el nodo si recorremos el nivel de izquierda a derecha y no se hubiese podado ningún elemento. De esta manera, si nos encontramos ante un nodo de decisión, sus hijos se encontrarían en el diccionario de la siguiente posición de la lista y sus índices de acceso serían el doble del índice de acceso de su padre o el doble del índice de acceso de su padre más 1, dependiendo de a cuál de los dos hijos queramos acceder. Por último, la descripción que se encuentra en en el diccionario para una clave de acceso es una tupla. Dicha tupla indica si el nodo es decisión o terminal. En el caso de ser un nodo de decisión, dispone de campos para el atributo y su valor de corte. En el caso de ser un nodo terminal, dispone de un campo que indica la clase con la que etiquetar la muestra.

5.3.7. Uso de Spark.

Este modelo, al requerir la evaluación independiente de múltiples nodos, como podremos comprobar posteriormente, no escala bien con la generación de árboles profundos o completos. Es por eso que, a la hora de integrar *Spark* en la solución, en vez de generar un único árbol, vamos a generar un *random forest*, es decir, vamos a subdividir la muestras de entrenamiento aleatoriamente de tal manera que cada partición de *Spark* entrenará un árbol y, una vez los árboles han sido entrenados, una muestra a clasificar será evaluada por todos los árboles. La clase de la muestra se corresponderá a la clase que ha sido seleccionada en el mayor número de árboles.

Esta solución nos permite, por un lado, reducir problemas de sobreajuste, y, por otro, conseguir precisiones competentes sin necesidad de generar árboles

completos u otros sistemas de poda más complejos y evitar los problemas de sincronización y comunicación que generaría el entrenamiento de un único árbol como que todos los nodos de *Spark* tendrían que tener una copia de las listas de atributos y, al terminar cada nivel de profundidad debería de plantearse una estrategia para reordenar las listas de atributos y volver a distribuir los cambios a todos los nodos.

Capítulo 6

Desarrollo de pruebas y análisis de resultados.

6.1. Entorno de pruebas.

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

- **GPU**: Zotac GeForce GTX 1060 AMP! Edition.

Características	Valor
Núcleos CUDA	1290
Frecuencia del procesador	1771 MHz
Frecuencia de la memoria	4004 Mhz
Memoria global total	6 GB DDR5
Bus de memoria	192-bit
Compute Capability	6.1
Número de hebras por bloque	1024
Dimensión máxima del bloque (x, y, z)	1024, 1024, 64
Dimensión máxima del "grid" (x, y, z)	2147483647, 65535, 65535
Número de registros por bloque	65536
Memoria compartida por bloque	49152 KB
Número de multiprocesadores	10
Modelo de driver CUDA	WDDM
Versión del driver CUDA	417.35
Versión del SDK CUDA	10.0

Cuadro 6.1: Características de la GPU NVIDIA GeForce GTX 1060 6 GB

- **Placa Base:** MSI B450M Bazooka.

- **Sistema Operativo:** Windows 10 Home 64 bits.
- **CPU:** AMD Ryzen 5 2600X.
- **RAM:** Kingston HyperX Fury Black DDR4 2400 MHz PC4-19200 8GB CL15.

6.2. Conjuntos de datos utilizados.

Durante la fase de desarrollo del mapa auto-organizado 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* [20]. 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.

Para evaluar el rendimiento de ambos modelos para conjuntos de *Big Data* hemos utilizado **SUSY** [21]. 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 diferenciar un proceso que genera partículas supersimétricas (*signal*) de otro proceso que no las genera (*background*). En el caso del mapa auto-organizado, la clase de salida es ignorada. De manera similar al anterior, los datos del conjunto fueron generados a partir de simulaciones de Monte Carlo.

6.3. Experimentos para evaluar el mapa auto-organizado.

6.3.1. Verificación de la implementación del modelo.

En el caso del mapa auto-organizado, tanto la versión como para CPU como para GPU ejecutan el mismo algoritmo, por lo que las métricas de interés durante las ejecuciones realizadas son el tiempo de ejecución y la ganancia. En primer lugar, durante la fase de desarrollo usamos el conjunto de las caras de *Olivetti*, que nos permitió comprobar de manera empírica y visual que los resultados obtenidos por el algoritmo son son correctos. En caso de funcionar correctamente, obtendríamos un conjunto de imágenes con la misma dimensión del mapa de neuronas, 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 este experimento, 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. El *RDD* de *Spark* que contiene las muestras de entrada es configurado para utilizar 10 particiones. Mientras que las versiones para CPU y GPU hacen exactamente lo mismo, utilizan métodos distintos para la generación de los pesos aleatorios iniciales. Por ello, para este experimento de verificación, tomamos también las dos medidas de calidad del mapa auto-organizado consideradas: el error de cuantificación y el error topográfico.

Figura 6.1: Imagen obtenida en el experimento para CPU del mapa auto-organizado.

En la figura 6.1, podemos observar los resultados obtenidos para la ejecución de este algoritmo sobre CPU. En ella, podemos observar que, personas con piel de color más oscuro se agrupan en la esquina superior izquierda, o que, en la fila inferior nos encontramos ante imágenes de la misma persona, donde en las 2 primeras imágenes el sujeto está mirando de lado y, en las siguientes, parece llevar gafas puestas, entre otros detalles. En este ejemplo, obtenemos un error de cuantificación de 6,57 y un error topográfico de 0,0325, tardando un total de 203,09 segundos en su ejecución.

En la figura 6.2, podemos observar los resultados obtenidos para la ejecución de este algoritmo sobre nuestro dispositivo *CUDA*. En ella, podemos observar como, personas que están claramente sonriendo se encuentran en la parte derecha de la penúltima fila, o en la esquina superior derecha, encontramos

Figura 6.2: Imagen obtenida en el experimento para GPU del mapa auto-organizado.

imágenes del mismo sujeto con gafas puestas. Para este ejemplo, obtenemos un error de cuantificación de 6,56 y un error topográfico de 0,0125, tardando un total de 281,01 segundos en su ejecución.

En este pequeño experimento, hemos podido comprobar visualmente que ambas implementaciones funcionan correctamente y proporcionan resultados similares, excepto en el tiempo de ejecución, y gran parte de los errores de implementación fueron detectados gracias a este experimento. El hecho de que la versión para GPU tarde más que la versión para CPU a que en cada una de las 10 particiones del *RDD* se evalúan tan sólo 40 muestras y, nuestro algoritmo, en cada iteración y para cada partición, ha de realizar transferencias de memoria entre host y dispositivo, añadiendo un *overhead*. Dado este número bajo de muestras, estamos invirtiendo más tiempo en realizar esas transferencias y lanzar los *kernels* que en los pocos cálculos necesarios. Conforme el número de muestras sea mayor, como veremos en el siguiente experimento, iremos obteniendo mejores resultados con la GPU.

6.3.2. 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 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 4 repeticiones del 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 tiempo de ejecución de nuestra implementación y la ganancia conseguida según vamos aumentando el número de muestras totales a evaluar. Nuestro RDD tendrá 10 particiones e iremos variando la cantidad de muestras totales de SUSY que vamos a procesar.

Nº de Muestras	***Tiempo CPU (s)***	***Tiempo GPU (s)***	***Ganancia***
500000	231,09	56,99	4,06
1000000	426,19	58,74	7,26
1500000	618,29	61,41	10,07
2000000	822,55	62,73	13,11
2500000	1017,45	66,22	15,36
3000000	1212,12	67,75	17,89
3500000	1398,09	67,14	20,83
4000000	1616,68	67,63	23,90
4500000	1788,50	68,45	26,13
5000000	1992,61	72,30	26,99

Cuadro 6.2: Tiempos promedios de ejecución y ganancias para el experimento del mapa auto-organizado sobre SUSY.

En la tabla 6.2, vemos las diferencias entre los tiempos promedios de 4 ejecuciones para CPU y 4 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 60-70 segundos (1 minuto). Sin embargo, la evolución de los tiempos para la CPU oscila entre los 231 segundos (casi 4 minutos) y 1992 segundos (33 minutos) por ejecución. Para una mejor visualización de estos resultados, planteamos la gráfica de la figura 6.3, en la que combinamos las gráficas de líneas para la evolución de los tiempos promedios con las ganancias obtenidas en una gráfico de barras.

En la gráfica planteada vemos de manera clara cómo, al aumentar el número de muestras, la implementación basada en *CUDA* y *Spark* es considerablemente más rápida que su homóloga para CPU. En el ejemplo más pequeño planteando, es decir, evaluar medio millón de muestras, en el que cada una de las 10 particiones del *RDD* evalúa 50000 muestras, la versión para CUDA es 4 veces más rápida que su homóloga para CPU. En el ejemplo más grande propuesto, es decir, evaluar 5 millones de muestras, el uso de CUDA nos ofrece un tiempo de ejecución casi 27 veces más rápido que la CPU.

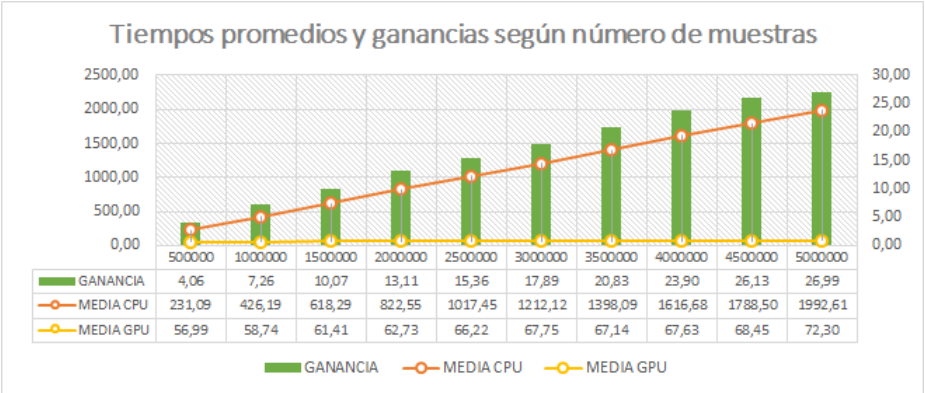

Figura 6.3: Gráfica con tiempos promedios y ganancias para SUSY.

6.3.3. Resultados de Nsight sobre la versión final del algoritmo.

Por último, analizamos en profundidad simulamos el entrenamiento del SOM durante una iteración con un ejemplo completamente aleatorio. El total de muestras a evaluar es de 1 millón, divididas en 10 particiones de 100000 muestras. El mapa de neuronas objetivo es de 10 filas por 10 columnas y la dimensión del problema a resolver es de 18 características. En la tabla 6.3, vemos los tiempos de ejecución de los *kernels* en este experimento.

		Tiempo			
Kernel	***Nº usos***	***mínimo***	***medio***	***máximo***	***total***
rand_weights	1	$6,62\ \mu s$	$6,62\ \mu s$	$6,62\ \mu s$	$6,62\ \mu s$
som_iter	10	$1,61\ ms$	$1,835\ ms$	$2,17\ ms$	$18,35\ ms$
finish_update	1	$37,38\ \mu s$	$37,38\ \mu s$	$37,38\ \mu s$	$37,38\ \mu s$

Cuadro 6.3: Tiempos de ejecución de los kernels en el experimento de profiling

Como cabía esperar, la mayor parte del tiempo se corresponde a la ejecución del *kernel som_iter*, que tarda un total de $18,35\ ms$. El kernel *rand_weigths* es el más rápido de todos y, aun así, es sólo invocado una vez en el algoritmo, independientemente del número de iteraciones, por lo que no tiene sentido centrarse en optimizarlo mientras se puedan hacer otras mejoras. El kernel *finish_update*, que será llamado tantas veces como iteraciones se realicen en el algoritmo ocupan el puesto intermedio, siendo 49 veces más rápido que una llamada al *kernel som_iter*. Además, el *kernel som_iter*, siempre será llamado las mismas veces que *finish_update* multiplicado por el número de

particiones del *RDD* por lo que, a ser posible, hemos de centrarnos en mejorar este *kernel*.

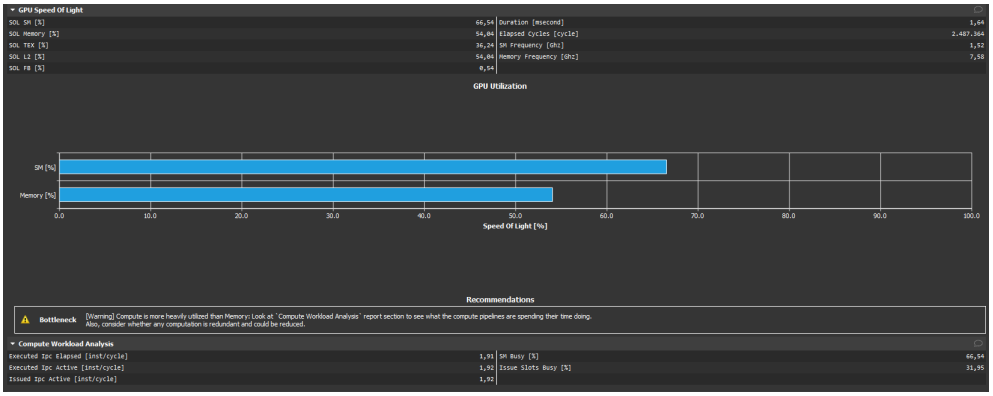

Figura 6.4: Speed of Light del kernel evaluado.

Un análisis más profundo del *kernel som_iter* con el profiler nos revela que, el cuello de botella ,se debe a los cálculos realizados (figura 6.4). La medida *"Speed Of Light (SOL)"* nos indica lo cerca que estamos de alcanzar el rendimiento teórico máximo de unidades de *hardware* durante la utilización del dispositivo CUDA en el *kernel*. Podemos observar que, en nuestro caso, estamos aprovechando un 66,54 % de la capacidad máxima de computación de nuestra GTX 1060.

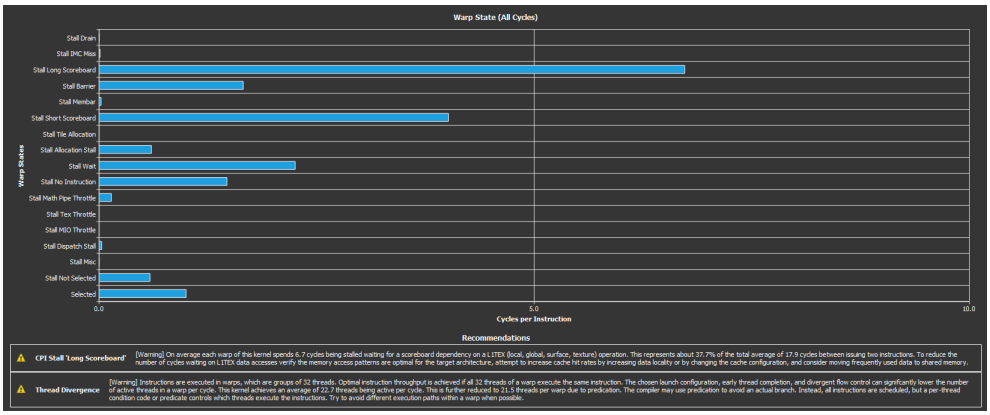

Figura 6.5: Análisis de los warps del kernel.

En la figura 6.5 vemos las dos razones principales que hacen que el *kernel* no funcione más rápido. Una de ellas es la cantidad de accesos a memoria dentro del dispositivo y la otra es la divergencia de las hebras, es decir, situaciones en las que las hebras quieren ejecutar diferentes instrucciones.

Todos los elementos dentro de un bloque que eran utilizados más de una vez fueron cargados en memoria compartida, por lo que, podemos intuir que este problema radica de conflictos que surjan del uso de las operaciones de suma atómica. Encontrar una alternativa mejor que la planteada para el cálculo de los pesos parciales debería de ser la prioridad a la hora de optimizar este algoritmo.

Por otro lado, el segundo problema que destacamos es la divergencia, que se debe principalmente a que, en este ejemplo, trabajamos con 128 hebras por bloque mientras que el número de neuronas es 100, por lo que uno de los *warps* utilizará tan sólo 4 de las 32 hebras. En un caso ideal, el tamaño del mapa tendría un número de neuronas que fuese potencia de 2 (al menos 32). Si tuviésemos garantizado que esto fuese a ocurrir, podríamos eliminar gran parte de los condicionales utilizados, ya que, no sería necesario comprobar si hay más hebras que neuronas, ni habría que rellenar distancias extras con infinito, ni tendríamos que realizar todas las comprobaciones en la reducción para tamaños de bloque superiores al número de neuronas.

6.4. Experimentos para evaluar el random forest.

Para la evaluación de los resultados del *random forest*, hemos utilizado el conjunto de datos SUSY. Para entrenar el *random forest*, se ha usado validación cruzada con 10 iteraciones, es decir, el conjunto de muestras se ha dividido en 10 subconjuntos de muestras del mismo tamaño; en cada iteración, 9 de esos subconjuntos han sido utilizados para entrenar el *random forest* y uno para evaluar los resultados; y, para finalizar, se ha realizado un promedio de los resultados. Para evaluar el modelo, hemos utilizado tres métricas: precisión, que indica el porcentaje de muestras que han sido correctamente clasificadas; el tiempo que tarda en entrenar el algoritmo; y la ganancia de la GPU a la CPU, o sea, cuántas veces es más rápida la tarjeta gráfica que el procesador.

Puesto que, tanto la versión para GPU como para CPU utilizan el mismo algoritmo pero, adaptado para las correspondientes arquitecturas, la precisión es un factor relevante a la hora de comprobar si la implementación de los algoritmos es correcta, ya que, en caso de una buena implementación, independientemente de haber usado GPU o CPU, obtendríamos precisiones idénticas, y estos resultados podemos compararlos con los obtenidos en aproximaciones similares en la literatura.

Además, durante el proceso de implementación, hemos utilizado otros conjuntos de datos, como Spambase [22] o *MAGIC* [23], para comprobar los resultados mientras utilizábamos un único árbol, cuyos resultados podemos ver en la siguiente tablas:

SPAMBASE 4601 muestras 57 atributos	***GPU***	***CPU (1 core)***	***GPU***	***CPU (1 core)***
Profundidad	***Tiempo (s)***		***Precisión (%)***	
4	0,06	0,19	87,77	87,77
5	0,1	0,24	89,65	89,65
6	0,14	0,32	91,04	91,04
7	0,21	0,42	91,82	91,82
8	0,3	0,55	92,11	92,11

Cuadro 6.4: Resultados de evaluar un árbol de decisión con validación cruzada con 10 iteraciones en SPAMBASE

MAGIC04 19020 muestras 10 atributos)	***GPU***	***CPU (1 core)***	***GPU***	***CPU (1 core)***
Profundidad	***Tiempo (s)***		***Precisión (%)***	
4	0,05	0,25	79,1	79,1
5	0,1	0,35	81,99	81,99
6	0,17	0,48	82,94	82,94
7	0,31	0,68	84,07	84,07
8	0,55	0,98	84,42	84,42

Cuadro 6.5: Resultados de evaluar un árbol de decisión con validación cruzada con 10 iteraciones en MAGIC04.

En ambas tablas podemos observar cómo la precisión obtenida en la versión para CPU como la versión para GPU es idéntica, garantizándonos que al menos ambas implementaciones, aunque hayan sido adaptadas para sus arquitecturas correspondientes, proporcionan los mismos resultados. Su elevado porcentaje de precisión, que se parecen a los obtenidos por aproximaciones similares presentes en la literatura llevan a pensar que la implementación es correcta. Por otro lado, aunque hayamos anotado el tiempo de ejecución en este experimento como referencia durante el proceso de desarrollo, hemos de recalcar que estamos comparando una versión del algoritmo en CUDA con una versión secuencial en la que sólo se utiliza uno de los núcleos de la

CPU, por lo que es normal que los tiempos de entrenamiento requeridos en la GPU sean inferiores a los que la CPU ha necesitado.

Para la evaluación del *random forest* con *SUSY* hemos generado un *random forest* con 12 árboles y hemos utilizado validación cruzada con 10 iteraciones sobre el total de los 5 millones de muestras disponibles en el conjunto de datos. El experimento ha sido realizado múltiples veces, variando la profundidad de 4 a 10. Además, el experimento ha sido repetido 5 veces, anotando los resultados promedios, que podemos observar en la tabla 6.6.

	GPU	***CPU***	***GPU vs CPU***	**GPU/CPU**
Profundidad	***Tiempo (s)***		***Ganancia***	**Precisión (%)**
4	25,58	27,64	1,08	75,82
5	26,06	31,71	1,22	76,99
6	26,20	35,30	1,35	77,44
7	26,45	38,19	1,44	78,33
8	27,34	40,87	1,49	78,67
9	29,97	45,85	1,53	79,02
10	33,63	48,66	1,45	79,25

Cuadro 6.6: Resultados de Random Forest para SUSY con 12 árboles

A diferencia del mapa auto-organizado, podemos observar que, en este caso, los resultados en este modelo se encuentran mucho más ajustados. Para poder realizar una interpretación adecuada de los tiempos de ejecución y la ganancia, hemos de tener en cuenta los siguientes factores:

- Dada la implementación realizada, la versión para GPU conseguirá mejores resultados conforme mayor sea el número de elementos a procesar.
- El número de árboles utilizados en el *random forest* va a influir en el tiempo de ejecución de la GPU. Conforme mayor sea el número de árboles utilizados, menor será el tamaño de elementos que tenemos garantizados que se procesen a la vez, si estamos usando una única GPU.
- Árboles muy profundos afectan de manera negativa a la GPU, debido a que, conforme pasemos al siguiente nivel de profundidad, vamos a tener un mayor número de nodos con un menor número de muestras cada uno o incluso parte de las muestras habrían sido eliminadas en alguna poda, que, limitando la capacidad de aprovechar la máxima capacidad de paralelismo posible.

En el ejemplo mostrado en la tabla 6.6, observamos que la versión para GPU nos ofrece una mejora del tiempo de ejecución al usar la GPU. En la profundidad 4, obtenemos el resultado que menos mejora nos ofrece con una diferencia de tan sólo 2 segundos. Mientras que la cantidad de datos sea lo suficientemente grande, una mayor profundidad implica más procesamiento de datos y vamos a obtener una mejor ganancia hasta la profundidad 9, en la que obtenemos la mejor ganancia, siendo la versión para GPU 1,53 veces más rápida que la correspondiente para CPU dándonos una ventaja de 15,88 segundos. En la profundidad 10, observamos cómo esa ganancia empieza a decaer bajando a 1,45 veces más rápido.

En el caso de la precisión, una mayor profundidad nos ayuda a obtener mejores resultados, partiendo de un 75,82 % en la profundidad 4 hasta un 79,25 % en la profundidad 10. . En comparación con otras aproximaciones de la literatura utilizadas para resolver este problema, la implementación realizada queda considerablemente por detrás del 85-87 % obtenido utilizado otros modelos [21].

Para concluir este análisis, podemos determinar que varios factores van a influir en qué situaciones va a ser mejor nuestra aproximación.

- Para considerar utilizar este modelo hemos de estar trabajando con bases de datos de una combinación de número de muestras y número de atributos considerablemente grandes.
- El número de árboles utilizados va a influir tanto en la precisión como en el tiempo de ejecución. Por regla generar, un mayor número de árboles va a mejorar la precisión pero va a reducir la velocidad de entrenamiento si se va a utilizar una única GPU. En el caso de disponer de un clúster, tener un número de GPUs igual al número de árboles nos permitiría obtener los mejores resultados, siempre y cuando el tiempo de comunicación entre los nodos del clúster no sea demasiado alto.
- Dependiendo del problema a resolver podremos variar el número de árboles o la profundidad que queremos usar para entrenar. Como veíamos antes, para casos más pequeños, como *Spambase* o *MAGIC04*, en los que utilizábamos un único árbol obteníamos resultandos competentes con la literatura en términos de precisión (92,11 % para Spambase y 84,42 % para MAGIC04, utilizando árboles de profundidad 8 en ambos casos).

6.4.1. Resultados del uso del profiler sobre la versión final del algoritmo.

En el caso del árbol de decisión desarrollado, el principal cuello de botella es la necesidad de lanzar una gran cantidad de *kernels* muy rápidos que no hemos sido capaces de compactar para evitar el *overhead* de cada lanzamiento de un kernel. Para evitarlo, hemos utilizado algunas técnicas para evitar este problema como el uso de *streams* concurrentes o la combinación de múltiples *kernels* (scan con cálculo del criterio de Gini y scan con el cálculo de las nuevas posiciones en la lista de atributos). Cabe también destacar, que otra alternativa a evaluar para solucionar este problema es el uso del paralelismo dinámico de *CUDA*, que permite lanzar un kernel desde otro, sin embargo, no tenemos acceso a esa característica usando *Python*.

Puesto que hay una alta variedad de *kernels*, que son invocados frecuentemente, vamos a observar la siguiente gráfica, en la que vemos el porcentaje del tiempo de ejecución que se dedica a cada tarea en una reproducción del experimento, con una base de datos de 100000 muestras aleatorias con 18 atributos y generando un árbol de profundidad 10.

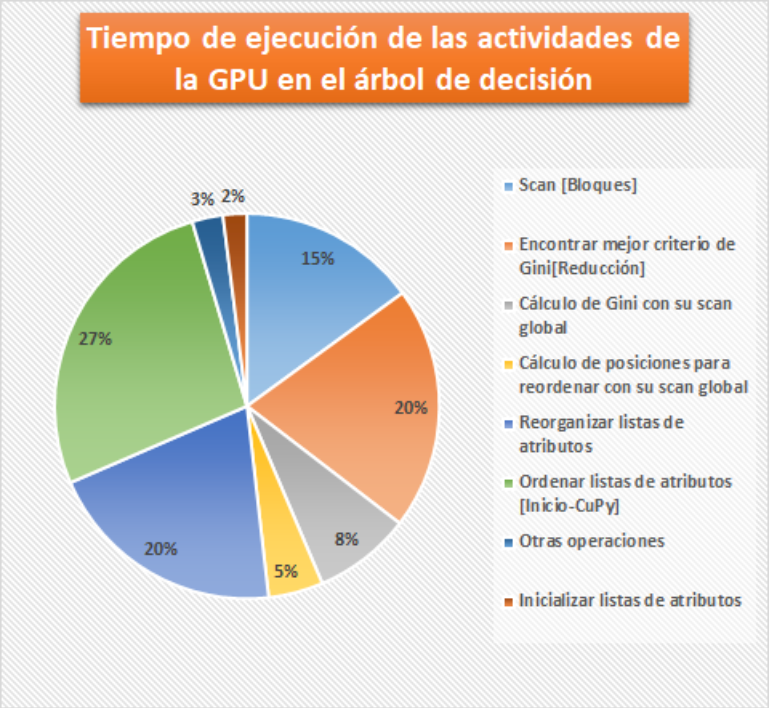

Figura 6.6: Análisis de los kernels ejecutados.

En primer lugar, la operación que más porcentaje de tiempo lleva es la ope-

ración más costosa, la organización de la listas de atributos inicial que toma un 27 % del tiempo total de ejecución de la GPU, correspondiente al algoritmo de ordenación de CuPy. Esta operación, que junto a la inicialización de la lista de atributos supone el inicio del algoritmo, acumulado un 29 % del tiempo usado. Una posible vía de mejora es evaluar si una implementación propia del algoritmo de ordenación nos proporcionaría resultados mejores. En caso de realizar una implementación propia, podríamos combinar con facilidad la inicialización de las listas de atributos con el algoritmo de ordenación.

En segundo lugar, tenemos un empate al 20 % entre la reducción utilizada para calcular el criterio de Gini y la reorganización de la listas de atributos. La operación de reducción es invocada para todos los nodos activos mientras que la reorganización de atributos es realizada una vez por nivel de profundidad.

La implementación de la reducción para este algoritmo sigue una estructura similar a la operación de *scan*. Ésta, hace uso de operaciones intrínsecas de *warps* y de operaciones atómicas para calcular el mínimo global entre los bloques. Otra alternativa probada fue el lanzamiento de otro kernel para el cálculo de los resultados globales, pero resultó ligeramente más lenta que el uso de las operaciones atómicas. Otra opción a evaluar, sería el uso del paralelismo dinámico, pero como comentamos previamente, no podemos acceder a esa característica desde *Numba*.

La reorganización de la listas de atributos es una operación de transferencia de memoria dentro del dispositivo. En primer lugar, los datos son movidos a *arrays* temporales y, una vez finalizado el proceso, se sobrescriben los datos en las nuevas posiciones. Dentro del dispositivo *CUDA*, estas transferencias de memoria son considerablemente rápidas, especialmente cuando los datos siguen algún patrón de adyacencia, como sucede al rellenar los arrays temporales. Sin embargo, en el caso de los accesos a memoria aleatorios, como ocurre en la fase de escritura, el tiempo de acceso va a ser mucho más lento, que queda manifestado en que, de ese 20 %, un 5,1 % se corresponde a la primera fase y el resto a la segunda.

Por último, vamos a analizar las tres operaciones relacionadas con la primitiva de *scan*, que son las siguientes en la gráfica. La operación de *scan* es usada dos veces por nodo activo, una primera vez para calcular el criterio de Gini y, una segunda vez, para calcular qué transferencias de memoria permiten mantener el orden de los atributos sin tener que volver a usar el algoritmo de ordenación, haciendo de ella la operación que es invocada el mayor número

de veces por ejecución del algoritmo. A diferencia del caso de la reducción, en vez de utilizar operaciones atómicas, usamos el lanzamiento de un nuevo *kernel* para realizar el cálculo global del *scan*. Este cambio se debe a que, dadas las necesidades de lanzar otros *kernels* para ejecutar el algoritmo sin paralelismo dinámico hemos podido combinar el cálculo del *scan* global con la operación posterior. La parte que realiza el cálculo de la primitiva *scan* para cada bloque representa un 15 % del tiempo de ejecución de la GPU mientras que el *kernel* que termina la primitiva y calcula el criterio de Gini conlleva un 8 % del tiempo y el cálculo de los nuevas posiciones para ordenar un 5 %. Puesto que, ambos *kernels* parten de la misma operación (realizar el *scan* global) podemos deducir que la complejidad aritmética del cálculo del criterio de Gini es superior a la del cálculo de las nuevas posiciones.

Aunque el análisis previo nos facilita saber qué *kernel* nos interesaría optimizar primero es importante observar un resultado más genérico obtenido en el *profile* nos indica que durante los 0,25 segundos que ha tardado en entrenarse el árbol sólo 0,1 segundos han sido usados para realizar los cálculos, proporcionado un porcentaje de utilización de la GPU del 40 %. Este problema deriva principalmente del *overhead* de tener que lanzar múltiples *kernels*, que viene dada por la necesidad de evaluar múltiples nodos independientes, que, además, han de ser sincronizados al terminar cada nivel de profundidad para reorganizar las listas de atributos y poder pasar al siguiente nivel.

Capítulo 7

Conclusiones y trabajos futuros.

En este trabajo, hemos desarrollado dos algoritmos de *Soft Computing*: el mapa auto-organizado de Kohonen y un árbol de decisión, realizado una adaptación de los mismos para dispositivos *CUDA*, y evaluando los resultados obtenidos con conjuntos de datos masivos mientras utilizamos el *framework Spark*, que nos ha llevado a convertir nuestro modelo que utiliza un único árbol de decisión a un *random forest*.

En el caso del mapa auto-organizado de Kohonen, la combinación de GPU y *Spark* proporciona resultados significativamente mejores que la versión para CPU, llegando a ser 27 veces más rápida la primera que la segunda en el mejor de los casos probados. Sin embargo, en el caso del *random forest*, la implementación utilizada, aunque proporciona ligeras mejoras en velocidad si el número de datos a procesar es lo suficientemente grande, no llega a las proporciones del primer caso. Esto se ha debido a que, en el primer caso, hemos afrontado un problema que podía ser resuelto de forma masivamente paralela mientras que, en el segundo, múltiples factores como, por ejemplo, la necesidad de procesar múltiples nodos de forma independiente, han limitado las posibilidades del dispositivo *CUDA*.

Por otro lado, el uso de *Python* y *Spark* ha resultado clave para poder desarrollar este proyecto de forma eficiente. *Spark* nos ha proporcionado herramientas para trabajar con bases de datos masivas, integrarlo con *CUDA* mediante Python y permitirnos realizar una implementación que podríamos llevar a un clúster con múltiples GPUs.

Las pruebas han sido realizadas sobre una única máquina y, aunque, en

ambas pruebas el uso de la GPU con bases de datos masivas ha resultado favorable, no podemos concluir que el uso de *CUDA* sea favorable siempre, aunque si podemos decir con certeza que, si el problema es masivamente paralelizable y se usa una base de datos lo suficientemente grande, el uso de la GPU será significativamente superior a los resultados obtenidos por la CPU.

Para terminar, vamos a comentar brevemente algunas vías de desarrollo por las que se podría ampliar el trabajo realizado:

- Añadir nuevos algoritmos de *Soft Computing* o más bases de datos sobre las que realizar pruebas.

- Realizar una comparativa de los resultados obtenidos en un rango de sistemas con especificaciones técnicas diferentes al utilizado.

- Realizar un análisis más profundo sobre las implicaciones de modificar ciertos parámetros utilizados en los algoritmos, como el número de particiones de *Spark*, el tamaño del mapa de Kohonen o la profundidad de los árboles.

- Realizar una comparativa de los resultados obtenidos entre clúster de GPU y clúster de CPU.

Bibliografía

[1] NVIDIA, *Procesamiento paralelo CUDA*. Consultado el 27 de abril de 2019. Disponible en `https://www.nvidia.es/object/cuda-parallel-computing-es.html`.

[2] 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, y I. Stoica, *Apache Spark: A Unified Engine for Big Data Processing*, vol. 59. New York, NY, USA: ACM, Octubre 2016. Disponible en `http://doi.acm.org/10.1145/2934664`.

[3] T. Kohonen, *The self-organizing map*, vol. 78, Proceedings of the IEEE. Septiembre 1990.

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

[5] T. Kohonen, *Self Organizing maps*, vol. 30, Springer Series in Information Science. 1995.

[6] J.-C. Fort, P. Letrémy, y M. Cottrell, *Advantages and drawbacks of the Batch Kohonen algorithm*. Enero 2002.

[7] F. Berzal, J. C. Cubero, F. Cuenca, y M. J. Martín Bautista, *On the quest for easy to understand splitting rules*, vol. 44. 2003.

[8] T. Oliphant, *Guide to NumPy*. Enero 2006.

[9] S. Kwan Lam, A. Pitrou, y S. Seibert, *Numba: a LLVM-based Python JIT compiler*. 11 2015.

[10] R. Okuta, Y. Unno, D. Nishino, S. Hido, y C. Loomis, *CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations*. 2017. Disponible en `http://learningsys.org/nips17/assets/papers/paper_16.pdf`.

[11] S. Vigna, *Further scramblings of Marsaglia's xorshift generators*, vol. 315. 2017. Disponible en `http://www.sciencedirect.com/science/article/pii/S0377042716305301`.

[12] F. Codevilla, S. Botelho, N. Duarte Filho, y J. Gaya, *Parallel High*

Dimensional Self Organizing Maps Using CUDA. Octubre 2012.

[13] H. Daneshpajouh, P. Delisle, J.-C. Boisson, M. Krajecki, y N. Zakaria, *Parallel Batch Self-Organizing Map on Graphics Processing Unit Using CUDA*. Enero 2018.

[14] W.-T. Lo, Y.-S. Chang, R.-K. Sheu, C.-C. Chiu, y S.-M. Yuan, *CUDT: a CUDA based decision tree algorithm*, vol. 2014. Julio 2014.

[15] D. Svatensson, *Implementing Streaming Parallel Decision Trees on Graphic Processing Units*. Junio 2018. Disponible en `http://www.diva-portal.se/smash/get/diva2:1220512/FULLTEXT02.pdf`.

[16] Y. Ben-Haim y E. Tom-Tov, *A Streaming Parallel Decision Tree Algorithm*, vol. 11. Febrero 2010.

[17] M. Harris, *Optimizing parallel reduction in CUDA*, vol. 21. Enero 2007.

[18] J. Shafer, R. Agrawal, y M. Mehta, *SPRINT: A scalable parallel classifier for data mining*. Agosto 2000.

[19] S. Sengupta, M. Harris, M. Garland, y J. Owens, *Efficient Parallel Scan Algorithms for GPUs*. Enero 2011.

[20] AT&T Laboratories Cambridge, *The Olivetti Faces dataset*. 1994, enlace Consultado el 6 de abril de 2019. Disponible en `https://scikit-learn.org/0.19/datasets/olivetti_faces.html`.

[21] P. Baldi, P. Sadowski, y D. Whiteson, *Searching for Exotic Particles in High-Energy Physics with Deep Learning*, vol. 5. Julio 2014.

[22] D. Dua y C. Graff, "Spambase, UCI Machine Learning Repository," Consultado el 20 de abril de 2019. Disponible en `https://archive.ics.uci.edu/ml/datasets/spambase`.

[23] D. Dua y C. Graff, *MAGIC, UCI Machine Learning Repository*. Consultado el 20 de abril de 2019. Disponible en `https://archive.ics.uci.edu/ml/datasets/magic+gamma+telescope`.