TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA.

Desarrollo e Implementación de modelos paralelos de Soft Computing en CUDA

Autor
David Criado Ramón

Directores
Manuel I. Capel Tuñón
María del Carmen Pegalajar Jiménez

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, junio de 2019

Desarrollo e Implementación de modelos paralelos de Soft Computing en CUDA

David Criado Ramón

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

Resumen

Poner aquí el resumen.

Project Title: Project Subtitle

David Criado Ramón

Keywords: Keyword1, Keyword2, Keyword3, ...

Abstract

Write here the abstract in English.

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

Fdo: David Criado Ramón

Granada a X de mes de 2019.

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

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

Informan:

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

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

Los directores:

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

Agradecimientos

A Rubén, por estar siempre apoyándome.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción y motivación.

1.1. 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, es muy utilizada y popular entre la comunidad científica.

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

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

El propósito de este trabajo de fin de grado es la implementación en CUDA de algunos de estos modelos de *Soft Computing* combinando *CUDA* el *framework* de computación en clúster *Spark* [2]. De esta manera, los algoritmos que se desarrollen podrán ser ejecutandos 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. Uno de ellos es el mapas auto-organizado de *Kohonen* [3], modelo que se basa en una red neuronal artifical no supervisada que preserva propiedades

topográficas y, ...

1.2. Estado del arte: trabajos relacionados.

La paralelización de los mapas auto-organizados de Kohonen ha sido previamente estudiada para su paralelización en CUDA.

En ***Parallel High Dimensional Self Organizing Maps Using CUDA*** Codevilla, Bothelo, Filho y Gaya [4] 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 realiza en 3 fases. Una primera en la que con un valor p arbitrario menor que el número de hebras por bloque que indica cuantos "pasos" debe realizar una hebra para el cálculo de la distancia euclídea, una reducción para encontrar la mejor distancia y una adaptación de pesos de neuronas basada en la dimensión del problema.

En ***Parallel Batch Self-Organizing Map on Graphics Processing Unit Using CUDA*** Daneshpajouh, Delisle, Boisson, Krajecki y Zakaria [5] 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 en la CPU.

1.3. Descripción oficial del proyecto.

En este proyecto se pretende que el alumno diseñe, desarrolle e Implemente modelos en pararelo asociados a tradicionales algoritmos de *Soft Somputing*. Para ello se utlizará 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.

1.4. 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

1.5. Planificación de tareas.

1.6. 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.

1.7. Estructura del documento.

- En el primer capítulo, **Introducción y motivación**, hemos comentado los própositos para la realización de este trabajo y el grado de consecución de los objetivos planteados.
- En el segundo capítulo, **Modelos de Soft Computing considerados**, explicamos los fundamentos teóricos de los algoritmos de *Soft Computing* que hemos decidido paralelizar.

- En el tercer capítulo, **Implementación**, comentamos el proceso de desarrollo seguido así como explicamos las soluciones finales implementadas y comentamos algunas de las alternativas y problemas que surgieron durante la realización de las implementaciones.
- En el cuarto capítulo, **Desarrollo de pruebas y análisis de resultados**, indicamos qué pruebas se han realizado, mostramos los resultados obtenidos y analizamos en profundidad las implicaciones de los mismos.
- En el último capítulo, **Conclusiones y trabajos futuros**, finalizamos el trabajo destacando las implicaciones más importantes de los resultados obtenidos y mostramos posibles alternativas para amplíar nuestro trabajo.

Capítulo 2

Modelos de Soft Computing considerados.

2.1. Mapas 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 funcionamiento del estudio 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 define son las siguientes:

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

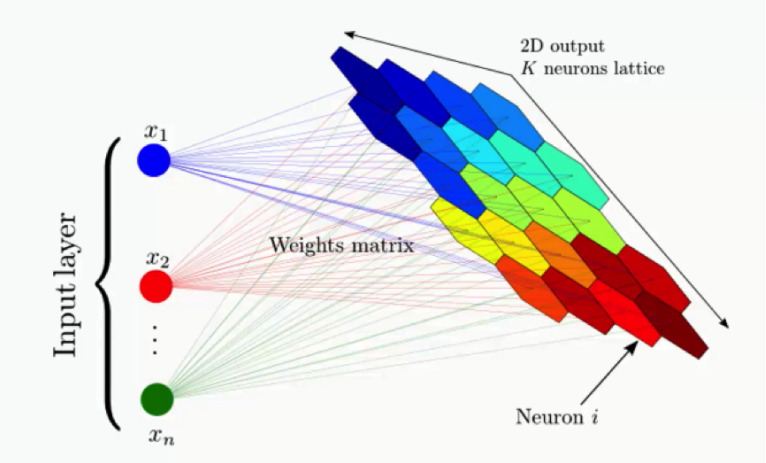

Figura 2.1: Esquema de una red neuronal de Kohonen.

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

- Es un algoritmo **no supervisado** capaz de encontrar patrones comunes basándose en los datos de la muestra de entrada sin necesidad de que cuando una muestra entre a la red se indique a qué categoría pertenece.
- Es un modelo **competitivo**. Cuando se recibe una muestra todas las neuronas compiten por ser activadas pero sólo la mejor será activada.

2.1.1. Proceso de entrenamiento.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Al tratar con una potencia con exponente negativo, un mayor valor absoluto de dicho exponente nos proporciona una valor de $\delta_f(i,j)$ menor. Por eso en el numerador se tiene en cuenta la distancia que hay entre la mejor neurona

y la neurona actual. En el denominador se utiliza un parámetro de control σque nos permite controlar la distancia que estamos considerando.

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

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

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

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

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

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

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

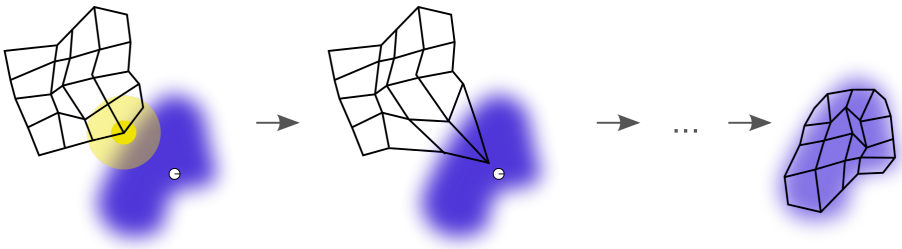

Figura 2.2: Proceso de entrenamiento de una red neuronal de Kohonen.

La figura 2.2 nos muestra, a grandes rasgos, un esquema del proceso de entrenamiento del mapa auto-organizado de Kohonen. 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 del 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 componentes son algunos de los modelos utilizados para visualizar el mapa auto-organizado.
- **Clasificación.** Una vez terminado el proceso de entrenamiento, puede asignarse etiquetas a cada uno de los nodos y resolver problemas de clasificación dependiendo de qué BMU se active.

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

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

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

El uso del modelo *batch* frente al modelo tradicional conlleva un intercambio de ventajas e inconvenientes [6] que podemos observar en la tabla 2.1.

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

Cuadro 2.1: Ventajas e inconvenientes de la versión batch.

2.1.4. Medidas de calidad.

Para medir la calidad de un mapa auto-organizado una vez entrenado podemos utilizar dos medidas:

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

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

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

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

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

Capítulo 3

Implementación.

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:

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

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

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

3.1.1. Python, NumPy, Numba.

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 [7] 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 [8] es un paquete para Python cuyo objetivo es la aceleración compilado fragmentos de código utilizando el compilador LLVM y dando la oportunidad de paralelizar código tanto para la CPU como para la GPU. En concreto, para las GPUs CUDA, proporciona al usuario un subconjunto de las características de CUDA con un nivel de abstracción mayor. Con eso no sólo conseguimos poder trabajar con CUDA desde Python sino también evitar, si lo deseamos, manejar 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.

3.1.2. Estructura de hebras, bloques y mallas.

El ***kernel*** es un fragmento de código especial, destino a ser ejecutado en el dispositivo, en el que se indica las instrucciones que 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
  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 el 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 que pueden cooperar entre sí y, 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

puede 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.

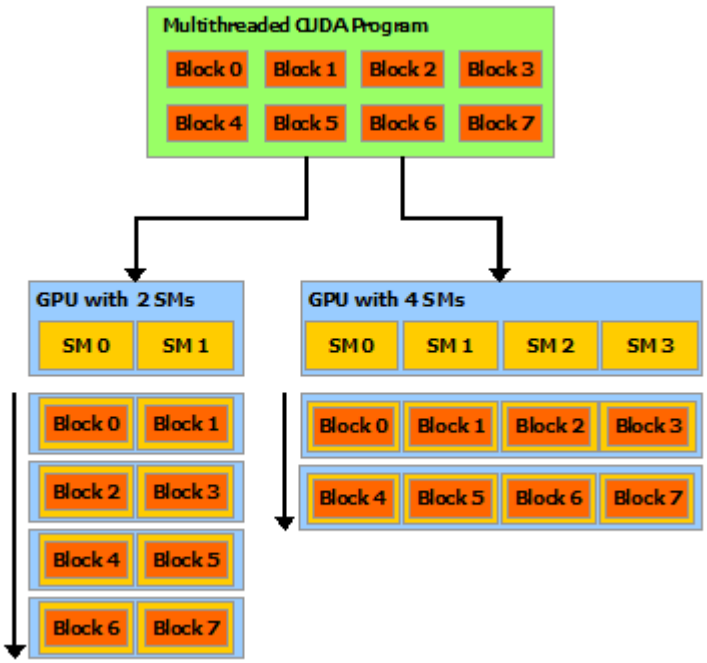

Figura 3.1: Distribución de bloques de un kernel en SMs.

Al lanzar un *kernel* hemos de utilizar al menos un bloque de N hebras. Además, en los casos unidimensionales, el número de hebras por bloque está limitado a un máximo que depende de la tarjeta gráfica en cuestión, 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*, cada bloque usaría 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 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 de datos entre *host* y dispositivo innecesaria.

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*.
d_array.copy_to_host()	Método para enviar d_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.

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 de que los cálculos previos que necesitamos se hayan realizado. Para evitar este tipo de dependencias, *CUDA* proporciona varios mecanismos para sincronizar las hebras usadas, 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 garan-

tizado que un *kernel* no será ejecutado hasta que el *kernel* anterior no haya 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.

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+* [9], 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)*.

En el código fuente 3.2, observamos cómo se inicializa un array de valores en coma flotante de 32 bits con pseudoaleatorios en *Numba CUDA*.

```
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.

3.1.6. La reducción y conflictos de bancos en memoria compartida.

Para finalizar la sección de *CUDA*, hablamos de una primitiva paralela muy utilizada en el mundo de la GPU, **la reducción**. El objetivo de esta primitiva es aplicar un operador binario, que cumpla la propiedad asociativa, a los elementos de un array, siendo el ejemplo más común realizar la sumatoria de todos los elementos de un array. Mientras que cuando trabajamos con una única hebra este algoritmo se realiza en $O(n)$ pasos, al realizarlo de forma paralela sólo se requieren $O(\log_2 n)$ pasos de todas las hebras. El algoritmo de la reducción simula hacer un recorrido en un árbol binario balanceado desde las hojas hasta la raíz. Los nodos hoja del último nivel se corresponden a los valores del array y el nodo raíz acabará almacenando el resultado final. En cada paso, una hebra guarda en el nodo padre el resultado de realizar la operación binaria sobre los valores de los dos hijos.

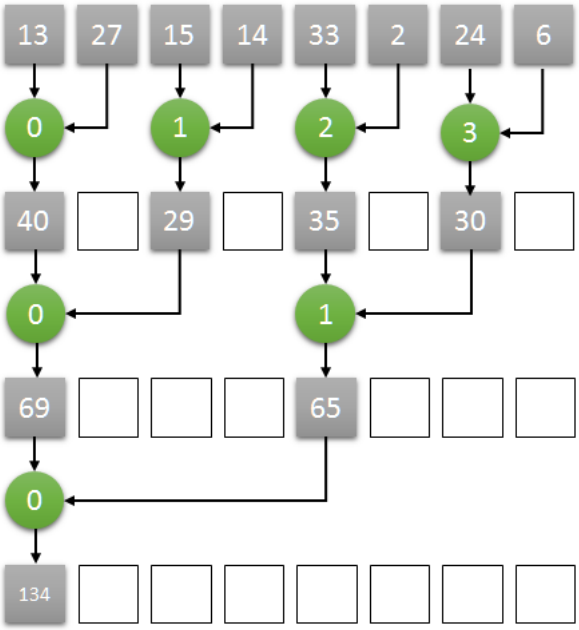

Figura 3.2: Una reducción paralela de una sumatoria.

En la figura 3.2, observamos cómo se aplicaría este proceso para realizar una sumatoria sobre un array de 8 elementos. En CUDA, este proceso se hace de una manera ligeramente distinta para evitar **conflictos de bancos de memoria compartida**. Un banco de memoria compartida, es una de las particiones de igual tamaño en las que se divide la memoria compartida. Habitualmente, el número de bancos es 32. Al trabajar con la memoria compartida, por defecto, palabras de 32 bits adyacentes son puestas en bancos

de memoria compartida consecutivos. Por ello, las direcciones de memoria de las posiciones 0, 32, 64, etc. del array se encontrarán en el banco 0; 1, 33, 65, etc. en el banco 1; y así sucesivamente. El conflicto surge en el caso en el que múltiples hebras de un *warp* intenten acceder a distintos elementos de un banco simultáneamente, ya que el acceso a memoria para cada hebra se secuencializa para cada hebra en conflicto. Si realizásemos la implementación de la forma propuesta en la figura 3.2, la hebra 0 leería de los bancos 0 y 1, la hebra 1 de los bancos 1 y 2, pero la hebra 16 también leería de los bancos 0 y 1, la hebra 17 de los bancos 2 y 3, y así hasta la hebra 32, generando un conflicto en cada banco y haciendo que una mitad del *warp* espere a la otra para realizar la siguiente operación. Para evitar esta situación, la hebra 0 trabaja con el dato en la posición 0 y la posición 512 (en el caso de trabajar con tamaños de bloque de 1024) haciendo que todos los datos de la hebra 0 se encuentren en bloque 0, la hebra 1 trabajaría con el dato de la posición 1 y el de la posición 513, que se encontraría en el bloque 1 y así para todos los bloques, evitando cualquier conflicto de bancos posible. En la figura 3.3 vemos cómo se realizaría este proceso.

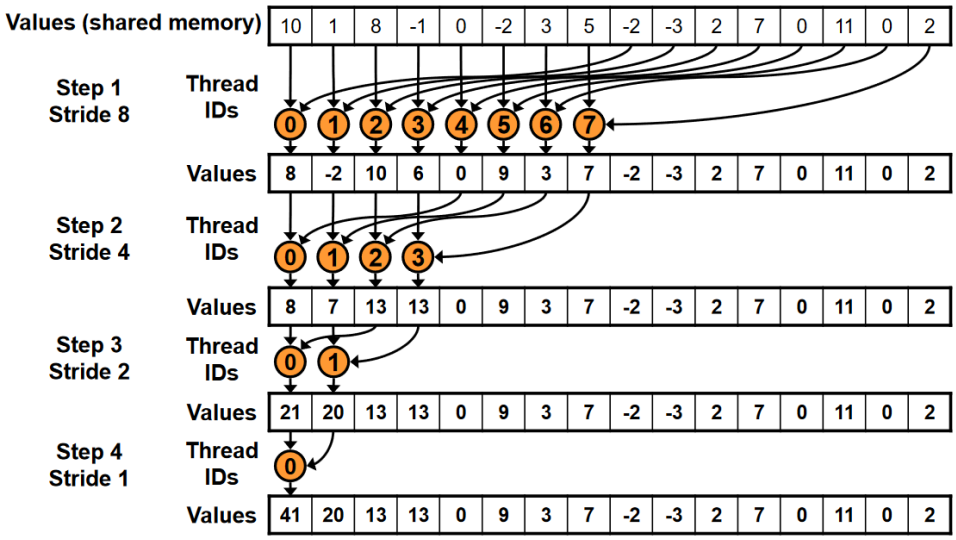

Figura 3.3: Reducción paralela de una sumatoria sin conflictos.

Por tanto, una implementación en *CUDA* de la reducción para un bloque haría lo siguiente:

- 1. Se declara un array de valores en memoria compartida. Si el tamaño del array a evaluar es potencia de 2 se toma como tamaño para el array en memoria compartida, en caso contrario, se toma la siguiente potencia de 2. El tamaño del array en memoria compartida será el número de hebras por bloque (*tpb*).

- 2. Cada hebra transfiere una posición del array en memoria global al array en memoria compartida. Si hay más hebras que valores tenía el array de memoria global, el resto de posiciones de memoria compartida se inicializan con el elemento neutro de la operación binaria, en el caso de la suma, el valor 0. Tras cargar los de datos se realiza una sincronización con **cuda.syncthreads()**.
- 3. Tomamos $n\_hebras = tpb/2$. Si el índice de la hebra (idx) es inferior a n_hebras se realiza la operación binaria entre las posiciones de memoria compartida idx e $idx + nhebras$. Tras cada paso es necesario usar **cuda.syncthreads()** para evitar riesgos de tipo *RAW* y n_hebras se reduce a la mitad. Este proceso es escrito manualmente en vez de usar un bucle para obtener el mejor rendimiento posible en tiempo de ejecución.
- 4. Una vez n_hebras es 32 o inferior no es necesario usar **cuda.syncthreads()** ya que, al estar todas las hebras en el mismo *warp*, se realizan todas las operaciones a la vez.
- 5. Cuando n_hebras es 1, en la posición de memoria compartida 0, se encuentra el resultado de la reducción sobre el bloque.

Si necesitamos trabajar con datos que no caben en un único bloque hemos de utilizar operaciones atómicas para combinar los resultados obtenidos o realizar múltiples lanzamientos del kernel desde la CPU. Podemos ver más detalles sobre cómo realizar una implementación eficiente de la reducción para CUDA en la referencia bibliográfica [10].

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. El *framework* fundamenta su arquitectura en el *RDD (Resilient Distributed DataSet)*, que es una estructura de datos de sólo lectura distribuida en un clúster de máquinas, mantenida durante toda la computación y con tolerancia a fallos. Además, proporciona otras herramientas de alto nivel como ML/MLib, una librería con algoritmos de *machine learning*.

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

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

3.3. 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.

3.4. 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*.

3.4.1. 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 trabjo 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 este 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 procediera. 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 el *clustering*.

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, 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*.

3.4.2. 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, como 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.

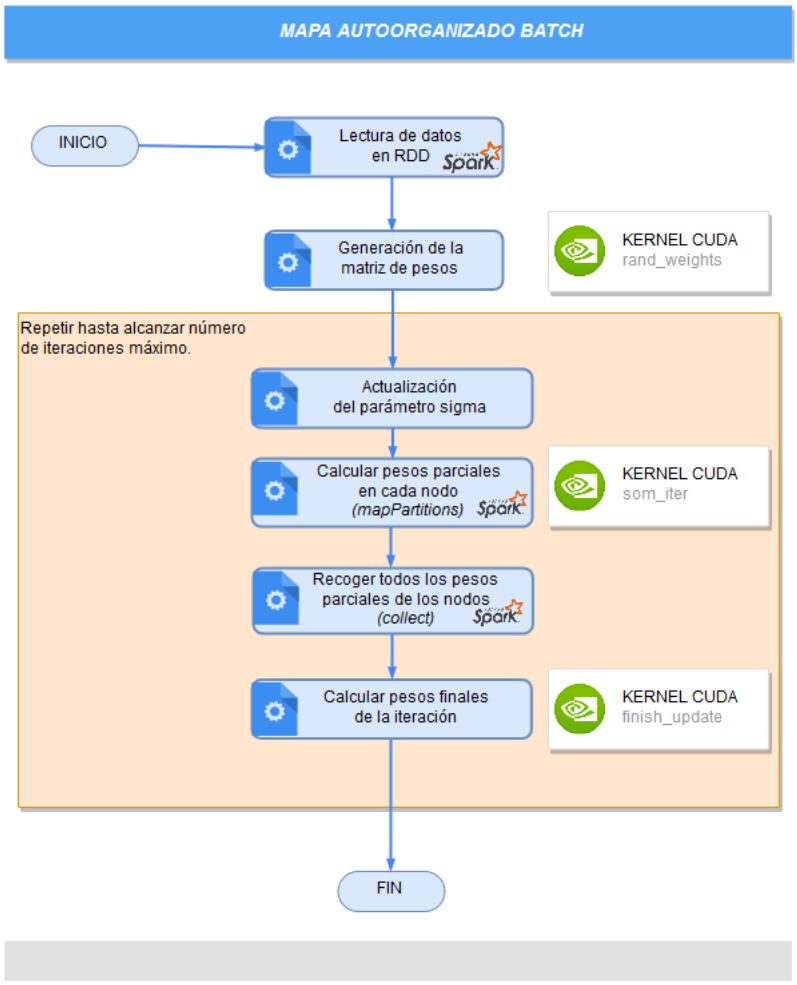

Figura 3.4: Diagrama de flujo del mapa auto-organizado desarrollado.

El algoritmo empieza en un único nodo de *Spark* utilizando el primer kernel

desarrollado, *rand_weights*, para 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 ecuación correspondiente.

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

- 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 costa 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 de memoria de dispositivo a *host* de los pesos parciales y, al inicio de la iteración, se haga la transferencia de *host* a 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 3.3: Uso de Spark para entrenar el mapa auto-organizado.

3.4.3. 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*.

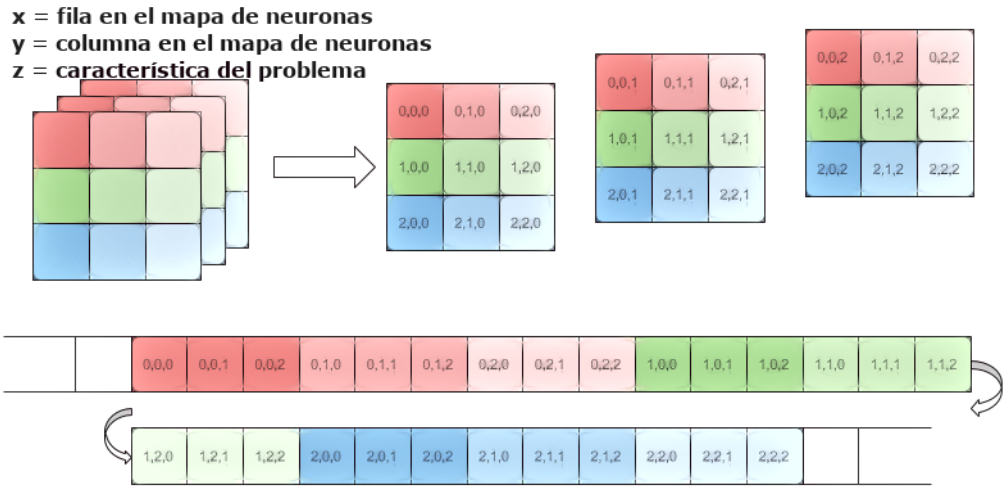

Figura 3.5: Representación de un array 3D como un array 1D row-major.

3.4.4. Kernels implementados.

3.4.4.1. Generación pseudoaleatoria de pesos de neuronas.

```
@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 3.4: Inicialización pseudoaleatoria de los pesos de las neuronas.

Este proceso (código fuente 3.4) 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 3.4), 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 3.4: Parámetros para el lanzamiento del kernel rand_weights.

3.4.4.2. 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 3.5: 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 3.3. 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 3.5: 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.

```python
@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 = cuda.blockIdx.x
    neuron_idx = cuda.threadIdx.x
    neuron_row = neuron_idx // ncols
    neuron_col = 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 3.6: 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**.

```
    # 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]
```

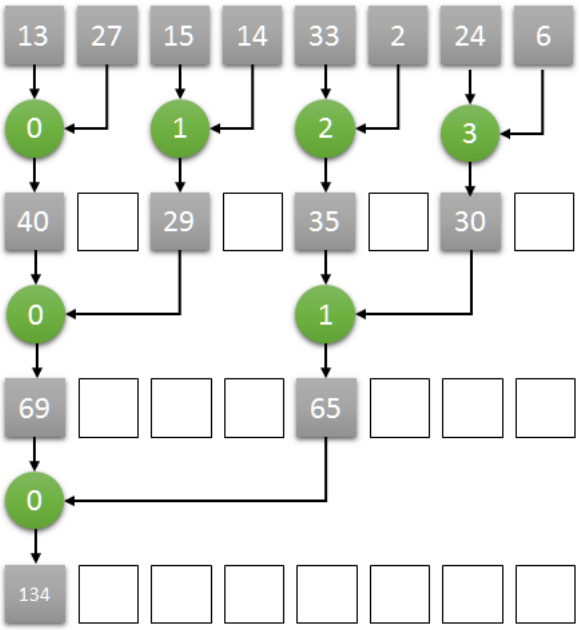

Figura 3.6: Una reducción paralela de una sumatoria en CUDA.

```
        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 3.7: 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 3.6), 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 [10].

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 3.8: 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 **opera-**

ciones 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.

3.4.4.3. 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 3.6: 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 3.9: Actualización final de la matriz de pesos.

Capítulo 4

Desarrollo de pruebas y análisis de resultados.

4.1. Entorno de pruebas.

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

- **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 4.1: 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.

4.2. 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* [11]. 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** [12]. 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.

4.3. Experimentos para evaluar el mapa auto-organizado.

4.3.1. 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 imagenes 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 autoorganizado consideradas: el error de cuantificación y el error topográfico.

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

En la figura 4.1, podemos observar los resultados obtenidos para la ejecución de este algoritmo sobre CPU. En ella, podemos observar 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 4.2, 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 imágenes del mismo sujeto con gafas puestas. Para este ejemplo obtenemos

Figura 4.2: Imagen obtenida en el experimento para GPU del mapa auto-organizado.

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 dipositivo, 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.

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

Posteriormente, para evaluar la capacidad del algoritmo ante un conjunto de mayores dimensiones, utilizamos SUSY. Para este experimento ignoramos las etiquetas de salida y utilizamos 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. Para todos los experimentos 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 4.2: Tiempos promedios de ejecución y ganancias para el experimento del mapa auto-organizado sobre SUSY.

En la tabla 4.2 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 4.3, 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.

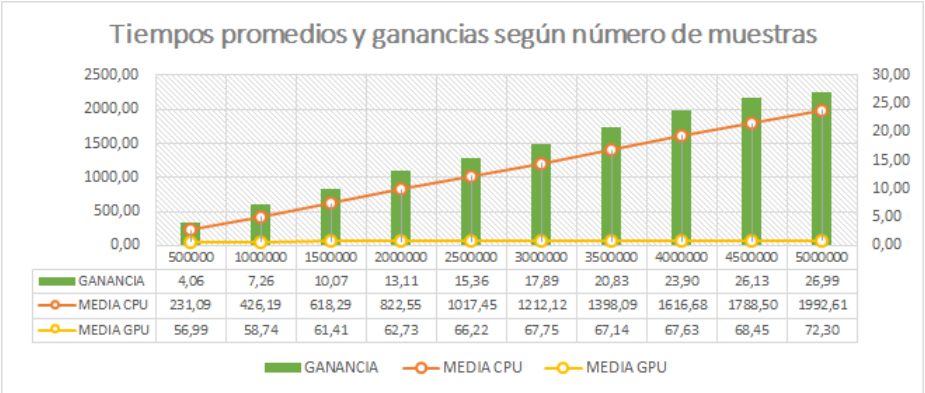

Figura 4.3: Gráfica con tiempos promedios y ganancias para SUSY.

4.3.3. 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 4.3, 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 4.3: 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*.

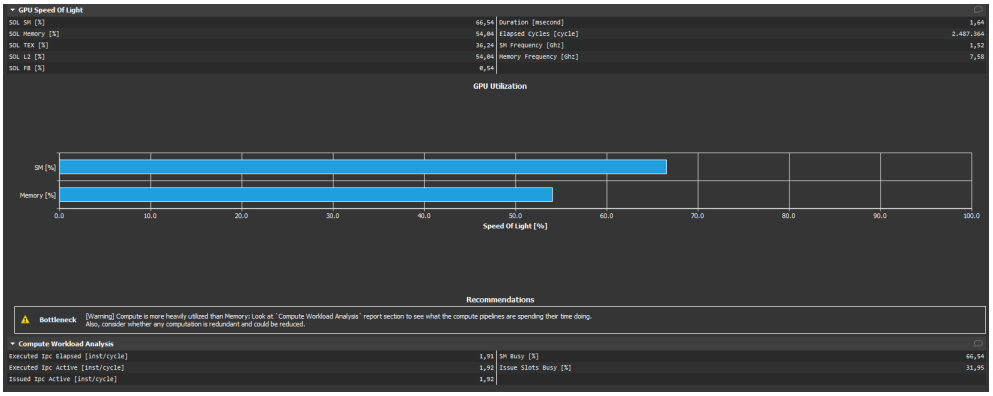

Figura 4.4: Speed of Light del kernel evaluado.

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 4.4). 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.

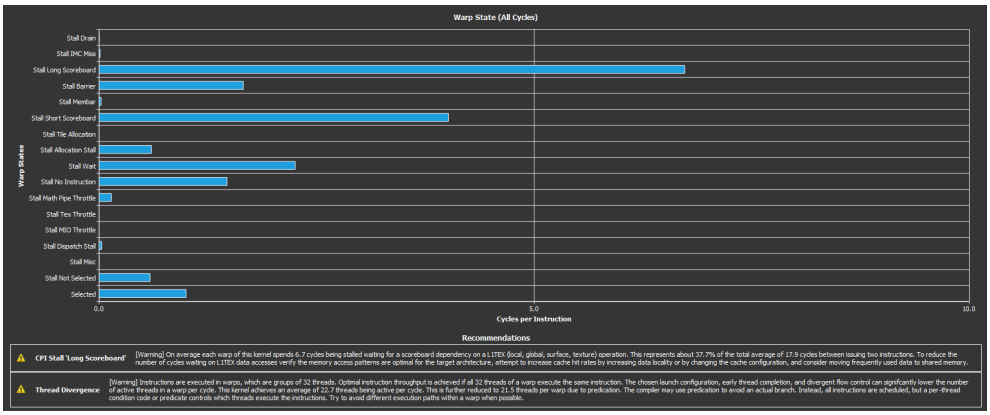

Figura 4.5: Análisis de los warps del kernel.

En la figura 4.5 vemos las dos razones principales que hacen que el *kernel* no funcione más rápido. Una de ellas es los 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.

Capítulo 5

Conclusiones y trabajos futuros.

Bibliografía

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

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

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

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

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

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

[7] T. Oliphant, *Guide to NumPy*, 01 2006.

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

[9] S. Vigna, "Further scramblings of marsaglia's xorshift generators," *Journal of Computational and Applied Mathematics*, vol. 315, pp. 175 – 181, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0377042716305301

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

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

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