

CLASIFICADOR PARALELO BASADO EN SOM

ARQUITECTURA DE LOS COMPUTADORES

**Ander Dorado, Julia de la Cruz, Juan Sáez, Daniel
Asensi, Laura Monroig, Alejandro Áyelo**

UNIVERSIDAD DE ALICANTE

Índice

1. Introducción	2
2. Entorno de desarrollo y requisitos del sistema	2
3. Arquitectura del SOM	2
4. Clasificación	3
5. Ficheros de entrada	3
6. Fichero de salida	4
6.1. Nuestra implementación	4
7. Transferencia de datos	5
7.1. Utilización de la memoria	5
7.2. La memoria compartida	6
7.3. Liberación de la memoria	6
7.4. Nuestra implementación	6
8. Kernel	8
8.1. Nuestra implementación	8
9. Arrays Multidimensionales	9
9.1. Distribución multidimensional de hilos	9
9.2. Nuestra implementación	9
10. Modificaciones de implementación	10
11. Comparativa de rendimiento	10

1. Introducción

En esta práctica implementaremos en CUDA una versión simplificada de un mapa auto-organizativo (Self Organizing Map). Se trata de una red neuronal que permite la clasificación de diferentes patrones de entrada. La red consta de dos modos: El modo entrenamiento que permite configurar la red y el modo de clasificación, que utiliza la configuración del entrenamiento para clasificar nuevos patrones de entrada. El SOM tiene la capacidad de presentar automáticamente un mapa dónde se puede observar una descripción del parecido entre los datos. El despliegue bidimensional tiene la propiedad de exhibir la información contenida en los datos de manera ordenada y resaltando las relaciones de similitud. El problema se centrará exclusivamente en la fase de clasificación.

2. Entorno de desarrollo y requisitos del sistema

Para la realización de la práctica hemos utilizado como lenguaje base C, acompañado de un conjunto de extensiones que permiten la implementación de algoritmos en CUDA. Las aplicaciones desarrolladas utilizan por una parte la CPU y por otra la GPU. La porción secuencial se ejecuta por parte de la CPU, por otro lado la porción paralela recae sobre la GPU, cada una con sus espacios de memoria distintos.

Antes de comenzar con la práctica debemos acondicionar nuestro dispositivo para que permita ejecutar aplicaciones de tipo '.cu'. Requisitos:

- **Sistema Operativo:** Microsoft Windows
- **Tarjeta gráfica NVIDIA:** Que posea cuda-enabled
- **Compilador C:** Microsoft Visual Studio
- **Software de CUDA:** Es necesario descargarse el software de CUDA correspondiente con tu tarjeta gráfica. Este software está dividido en CUDA Toolkit (contiene el compilador y se encarga de separar el código destinado al host y el destinado al device) y CUDA SDK (Software Development Kit)

3. Arquitectura del SOM

El SOM consiste en una colección de nodos que contiene un vector de pesos de la misma dimensión que la de los patrones de entrada. Además cada nodo (neurona) contendrá una etiqueta que será la salida del clasificador. Un modelo SOM se compone por dos capas de neuronas.

- **La capa de entrada**, formada por N neuronas, una por cada variable de entrada, se encarga de recibir y transmitir a la capa de salida la información procedente del exterior.
- **La capa de salida**, formada por M neuronas es la encargada de procesar la información y formar el mapa de rasgos.

La información se propaga desde la capa de entrada hacia la de salida. Conectando cada neurona de entrada con cada una de las neuronas de salida j mediante un peso w_{ji} . Por lo tanto las neuronas de salida están asociadas a un vector de pesos W_j o vector de referencia. El SOM define una proyección desde un espacio de datos en alta dimensión a un mapa bidimensional de neuronas.

Entre las neuronas de la capa de salida existen conexiones laterales de excitación e inhibición implícitas y aunque no estén conectadas cada una de ellas tendrá influencia sobre las adyacentes. Esta relación es más fuerte cuando la distancia física entre dos neuronas es más pequeña, es decir, si más cerca están, mayor es la interacción. Esto se consigue a través de un proceso de competición entre las neuronas y de la aplicación de una función denominada de vecindad, que produce la topología o estructura del mapa. Las topologías más frecuentes son la rectangular y la hexagonal.

Permanecen fijo la topología y el número de neuronas permanece. El número de neuronas determina la suavidad de la proyección que influye en el ajuste y capacidad de generalización del SOM. En la fase de entrenamiento, el SOM forma una red elástica que se pliega dentro de la nube de datos originales.

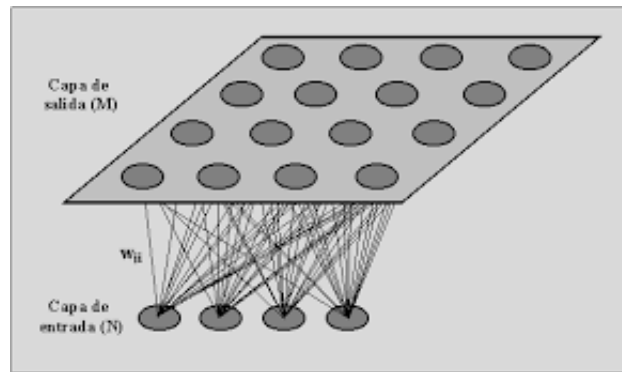


Figura 1: Organización de las neuronas

4. Clasificación

El SOM compara el patrón de entrada con cada una de las neuronas del mapa para establecer qué neurona es la ganadora. La ganadora es aquella cuya suma de distancias euclídeas entre el patrón y las neuronas de su vecindad, ella misma incluida, sea la menor en el mapa.

Pseudocódigo:

```

para toda N de las neuronas del mapa
    distancia(N) = euclidea (N,P);
    para toda neurona V de la vecindad de N
        distancia(N) = distancia(N) + euclidea(V,P)
    fin para
fin para
ganadora = min(distancia)

```

5. Ficheros de entrada

Al programa se le proporcionan la lectura de dos ficheros de entrada. Uno (Estructura.som) contiene la estructura del mapa, es decir, en él se indicará la anchura y el alto, además de la dimensiones de los pesos de cada neurona. Por ejemplo, un mapa 3x3 con un vector de pesos de dimensión 3 sería:

Alto: 3

Ancho: 3

Dimensión: 3

N1,1: 43.5, 53.5, 48.6

L1,1: 1

N1,2: 64.9, 37.5, 59.6

L1,2: 2

N1,3: 76.7, 65.8, 75.4

L1,3: 3

N2,1: 45.6, 34.5, 64.5

L2,1: 4

N2,2: 75.5, 54.5, 34.5

L2,2: 1

N2,3: 45.3, 64.6, 54.6

L2,3: 3

N3,1: 54.6, 57.5, 34.6

L3,1: 1

N3,2: 74.6 52.5, 43.6

L3,2: 2

N3,3: 54.7 83.6 65.7

L3,3: 4

El otro fichero de entrada (patrones.pat), contiene los patrones de entrada. En éste se indica el número de patrones y la dimensión. Por ejemplo:

Número: 4

Dimensión: 3

P1: 76.5, 64.3, 53.6

P2: 85.6, 43.5, 23.5

P3: 56.4, 34.6, 12.5

P3: 95.6, 45.3, 35.6

6. Fichero de salida

La salida se escribirá en fichero llamado salida.txt que contendrá el numero de etiquetas y el valor asociado a cada una de estas.

6.1. Nuestra implementación

Función generar salida: La función generar salida es llamada desde *runTest()* una vez ejecutado *ClasificacionSOMGPU()*

Función generar salida:

```
void
generarSalida(int* labels)
{
    char* fichero = "salida.txt";
    FILE* fpin;          /* Fichero */
    int label;
    float pesos;
    /* Apertura de Fichero */
    if ((fpin = fopen(fichero, "w")) == NULL) return;
    /* Lectura de cabecera */
    if (fprintf(fpin, "Numero: %d\n", TAM) < 0) return;
    if (feof(fpin)) return ;
}
```

```

for (int i = 0; i < TAM; i++)
{
    fprintf(fpin, "%d \n", (char*)resultado[i]);
}

fclose(fpin);
}

```

7. Transferencia de datos

7.1. Utilización de la memoria

El modelo de memoria utilizado en CUDA se compone de un *Host* y un *Device*, que poseen un espacio de memoria propio. El *kernel* solo podrá operar en el espacio de la memoria del dispositivo, por lo que se necesitarán funciones específicas para reservar y liberar dicha memoria, así como funciones para la transferencia de información entre los dos componentes anteriormente mencionados.

Las zonas accesibles desde el *Host* son la **memoria global** y la **memoria constante** y desde estas zonas de memoria el *kernel* puede transferir datos al resto de niveles, además los *hilos* podrán acceder a ambas zonas de la memoria, pero solamente dos hilos del mismo bloque podrán acceder a una zona de memoria compartida.

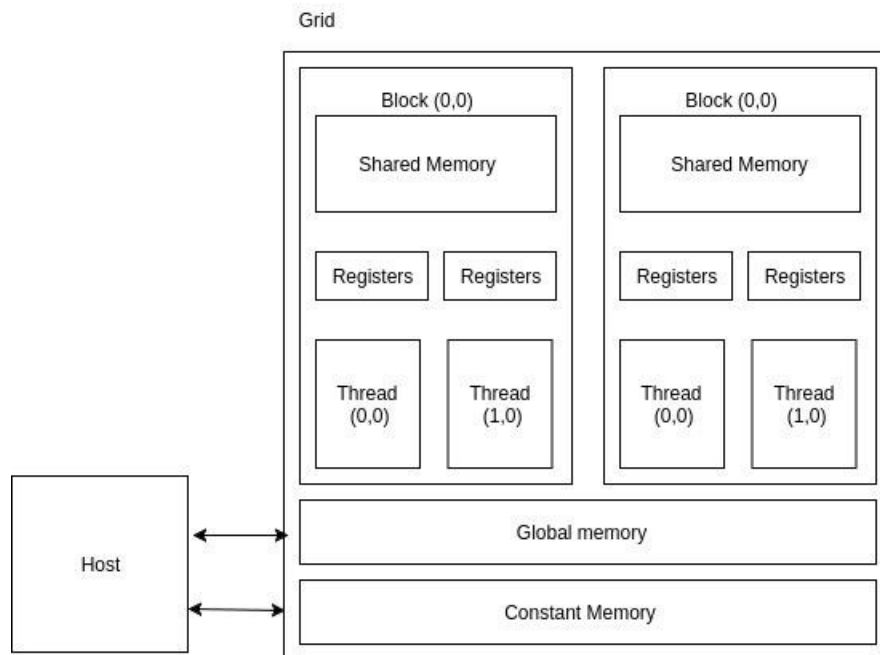


Figura 2: Representación de memoria de bloque, Host y Grid

7.2. La memoria compartida

Para poder reservar los espacios de **memoria global** en nuestra clasificación de SOM utilizaremos

```
cudaMalloc(void **devPtr, size_t size);
```

El primer argumento (*devPtr*) será la dirección del puntero a memoria donde queramos almacenar los datos, el segundo elemento (*size*), será la cantidad de *bytes* de que desearemos reservar dentro de la tarjeta gráfica. Una vez reservado este espacio deberemos mover los datos a almacenar desde nuestra CPU a la GPU, para ello utilizaremos **cudaMemcpy()** con la correspondiente sintaxis:

```
cudaMemcpy(void *dst, void *src, size_t count, cudaMemcpyKind kind);
```

Donde como los nombres de los parámetros indican, (*dst*) será un puntero al destino, (*src*) un puntero al origen, (*count*) la cantidad de bytes a copiar y (*kind*) el tipo de transferencia empleada que podrá ser de los siguientes tipos:

Tipo de transferencia	Sentido de la transferencia
cudaMemcpyHostToHost	host → host
cudaMemcpyHostToDevice	host → device
cudaMemcpyDeviceToHost	device → host
cudaMemcpyDeviceToDevice	device → device

Figura 3: Posibles transferencias de datos en CUDA

7.3. Liberación de la memoria

Con la siguiente llamada liberaremos el espacio de memoria apuntado por (*devPtr*), devuelto por la función previa de **cudaMalloc()**

Liberación de memoria en cuda:

```
cudaFree(void *devPtr);
```

7.4. Nuestra implementación

Transferencia de datos:

```
void
datosAccesiblesGPU(int tamArray, SOMNeurona aux, float* pesos, int* labels, PatronesCUDA
    &d_Patrones, int * d_EtiquetaGPU)
{
    cudaMalloc(&d_EtiquetaGPU, Patrones.Cantidad * sizeof(int));
    cudaMemcpy(EtiquetaGPU, d_EtiquetaGPU, Patrones.Cantidad * sizeof(int),
        cudaMemcpyDeviceToHost);

    pesosNeurona(pesos); // Array de los pesos de cada neurona

    cudaMalloc(&aux.pesos, tamArray); // Cargamos la memoria
    cudaMemcpy(aux.pesos, pesos, sizeof(pesos), cudaMemcpyHostToDevice);

    etiquetaNeurona(labels); // Array de salidas del clasificador
```

```

    cudaMalloc(&aux.label, (sizeof(int) * SOM.Alto * SOM.Ancho));
    cudaMemcpy(aux.label, labels, (sizeof(int) * SOM.Alto * SOM.Ancho), cudaMemcpyHostToDevice);

    d_Patrones.Cantidad = Patrones.Cantidad;
    d_Patrones.Dimension = Patrones.Dimension;

    float* pesosPatron = (float*)malloc(Patrones.Cantidad * Patrones.Dimension * TAMINT);
    pesoP(pesosPatron);

    cudaMalloc(&d_Patrones.Pesos, Patrones.Cantidad * Patrones.Dimension * TAMINT);
    cudaMemcpy(d_Patrones.Pesos, pesosPatron, tamArray, cudaMemcpyHostToDevice);
}

```

Funciones auxiliares transferencia de datos:

```

int bytesArray()
{
    int tamArray = 0;
    for (int x = 0; x < SOM.Alto; x++)
        for (int y = 0; y < SOM.Ancho; y++)
            tamArray += sizeof(SOM.Neurona[x][y].pesos);
    return tamArray;
}

void
pesosNeurona(float *pesos)
{
    for (int i = 0; i < SOM.Ancho; i++)
        for (int j = 0; j < SOM.Alto; j++)
            for (int k = 0; k < sizeof(SOM.Neurona[i][j].pesos) / TAMINT; k++)
                pesos[obtenerPosicion3D(i, j, k)] = SOM.Neurona[i][j].pesos[k];
}

void
etiquetaNeurona(int* labels)
{
    for (int i = 0; i < SOM.Ancho; i++)
        for (int j = 0; j < SOM.Alto; j++)
            labels[obtenerPosicion2D(i, j)] = SOM.Neurona[i][j].label;
}

void
pesoP(float* pesosP)
{
    for (int i = 0; i < Patrones.Cantidad; i++)
        for (int j = 0; j < Patrones.Dimension; j++)
            pesosP[obtenerPosicion2D(i, j)] = Patrones.Pesos[i][j];
}

```

8. Kernel

Los dispositivos CUDA aprovechan los paralelismos en los datos para acelerar las aplicaciones. Para ello, el *device* trabaja como un coprocesador del *host*, dividiendo el cálculo entre la CPU y la GPU. La ejecución comienza en el *host* y al invocar un Kernel se traslada al *device*, donde con tal de aprovechar los paralelismos se generan un gran número de hilos. Cuando los procesos en todos los hilos del kernel finalizan, la ejecución continúa en el *host* otra vez. El compilador de Nvidia será el encargado de separar el código del *host* con respecto al del *device*.

Por eso, es necesario indicar en la declaración de las funciones dónde se deben ejecutar en la CPU o en la GPU.

	Ejecutable en:	Invocable desde:
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

Figura 4: Especificadores de tipo para funciones

En cuanto a las llamadas de los kernels por parte de las funciones ejecutadas en la GPU, seguirán la siguiente sintaxis:

```
function <<<grid, blocks>>> (args...);
```

8.1. Nuestra implementación

Llamada al kernel:

```
codigoParelizable <<<grid, BLOCKS>>> (EtiquetaGPU, aux, d_Patrones);
```

Funcion codigoParelizable:

```
__global__  
void  
codigoParelizable(int* d_EtiquetaGPU, SOMNeurona d_SOMNeurona, PatronesCUDA d_PatronesCUDA) {  
  
    float distancia = 0;  
    float distanciaMenor = MAXDIST;  
    int np = threadIdx.x + blockIdx.x * blockDim.x;  
  
    int height = d_SOMNeurona.Alto;  
    int width = d_SOMNeurona.Ancho;  
  
    if (np < d_PatronesCUDA.Cantidad) {  
        distanciaMenor = MAXDIST;  
        for (int y = 0; y < d_SOMNeurona.Alto; y++) {  
            for (int x = 0; x < d_SOMNeurona.Ancho; x++) {  
                distancia = 0;  
                if (y >= 0 && y < d_SOMNeurona.Alto && x >= 0 && x < d_SOMNeurona.Ancho) {  
                    for (int i = 0; i < d_PatronesCUDA.Dimension; i++) {  
                        distancia += abs(d_SOMNeurona.pesos[y + x * height + i * height * width] -  
                                      d_PatronesCUDA.Pesos[np + i * d_PatronesCUDA.Cantidad]);  
                    }  
                    distancia = distancia / d_PatronesCUDA.Dimension;  
                }  
            }  
        }  
    }  
}
```

```

        for (int vy = -1; vy < 2; vy++)
            for (int vx = -1; vx < 2; vx++)
                if (vx != 0 && vy != 0)
                    if (y >= 0 && y < d_SOMNeurona.Alto && x >= 0 && x < d_SOMNeurona.Ancho)
                        for (int i = 0; i < d_PatronesCUDA.Dimension; i++)
                            distancia += abs(d_SOMNeurona.pesos[y + vy + (x + vx) * height +
                                i * height * width] - d_PatronesCUDA.Pesos[np + i *
                                    d_PatronesCUDA.Cantidad]);

                    distancia = distancia / d_PatronesCUDA.Dimension;

                if (distancia < distanciaMenor) {
                    distanciaMenor = distancia;
                    d_EtiquetaGPU[np] = d_SOMNeurona.label[y + x * d_SOMNeurona.Alto];
                }
            }
        }
    }
}

```

9. Arrays Multidimensionales

9.1. Distribución multidimensional de hilos

Hay situaciones donde es mejor distribuir los hilos de manera coherente al problema que se pretende solucionar. En nuestro caso operamos con matrices y por ello se distribuirían los hilos a lo largo del eje x y del eje y. El código para lanzar el kernel es exactamente igual pero las variables *blocks* y *threads* en este caso serán tridimensionales.

Debido a que `cudaMalloc()` es un **espacio lineal** el acceso a datos hay que hacerlo mediante un índice lineal a partir de los índices correspondientes a los ejes x e y. Para ello utilizaremos funciones auxiliares `__device__`.

9.2. Nuestra implementación

Funciones auxiliares índice:

```

int obtenerPosicion3D(int x, int y, int z) {
    return x*y*SOM.Alto+z*SOM.Alto*SOM.Ancho;
}

__device__
int obtenerPosicion3D(int x, int y, int z, int height, int width) {
    return x*y*height+z*height*width;
}

int obtenerPosicion2D(int x, int y) {
    return x*y*SOM.Ancho;
}

__device__
int obtenerPosicion2D(int x, int y, int height) {
    return x*y*height;
}

```

Blocks y threads:

```
const dim3 BLOCKS = 24;  
dim3 grid((Patrones.Cantidad + (BLOCKS.x - 1)) / BLOCKS.x);
```

10. Modificaciones de implementación

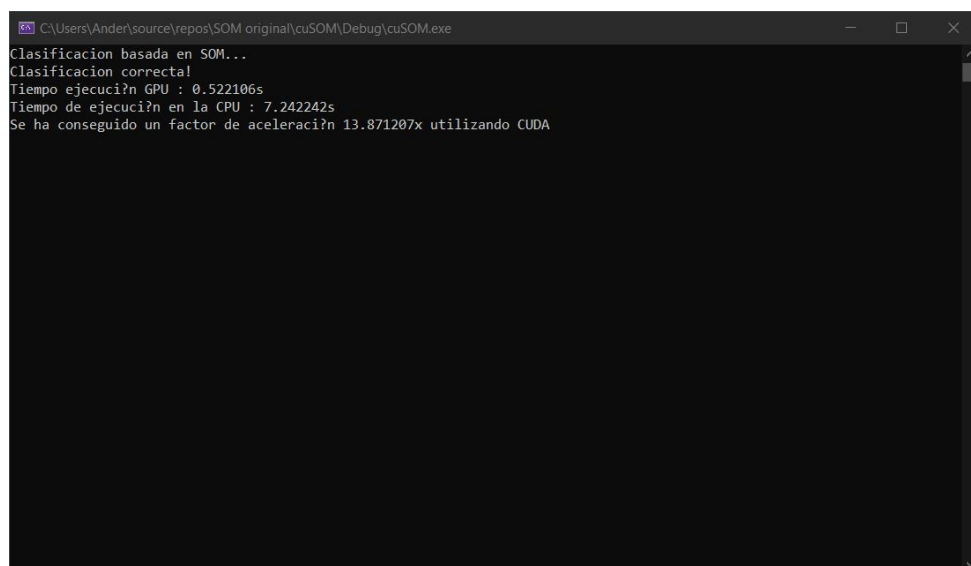
Debido a que no es posible copiar arrays de más de una dimensión en CUDA se ha decidido usar arrays de una sola dimensión. Para ello hemos implementado una nueva estructura similar a la dada pero con arrays unidimensionales.

Estructuras auxiliares (clasificacionSOM.h):

```
struct sSOMNeuronaCUDA  
{  
    int Ancho;  
    int Alto;  
    int Dimension;  
    float *pesos;  
    int *label;  
};  
  
typedef struct sSOMNeuronaCUDA SOMNeurona;  
  
struct sPatronesCUDA  
{  
    int Cantidad;  
    int Dimension;  
    float *Pesos;  
};  
  
typedef struct sPatronesCUDA PatronesCUDA;
```

11. Comparativa de rendimiento

Gracias a la arquitectura CUDA de Nvidia podemos acelerar el programa. El tiempo de ejecución en la CPU es de 7.24 segundos frente a los 0.5 segundos con la arquitectura CUDA. Hemos obtenido un factor de aceleración del x13.



```
C:\Users\Ander\source\repos\SOM original\cuSOM\Debug\cuSOM.exe  
Clasificacion basada en SOM...  
Clasificacion correcta!  
Tiempo ejecuci?n GPU : 0.522106s  
Tiempo de ejecuci?n en la CPU : 7.242242s  
Se ha conseguido un factor de aceleraci?n 13.871207x utilizando CUDA
```

Figura 5: Resultados de rendimiento