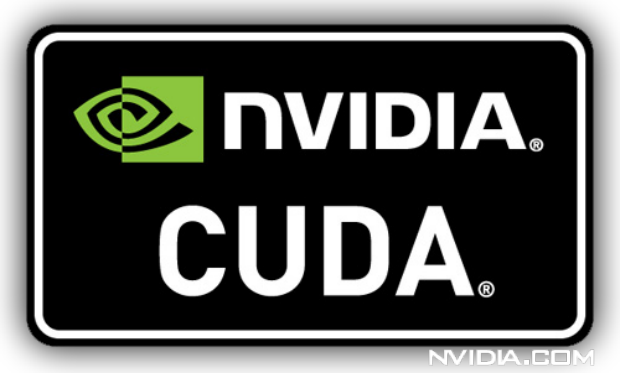
Programación paralela en CPU y GPU: Perceptrón multicapa para problemas de clasificación

Arquitecturas Paralelas – Grado de Ingeniería Informática

Universidad de Córdoba, EPSC

2020/2021



Trabajo realizado por:

-Antonio Gómez Giménez (i72gogia@uco.es)

-Rafael Hormigo Cabello (i72hocar@uco.es)

Índice:

1.- Descripción del problema

En este trabajo nos centraremos en la paralelización de redes neuronales *"perceptrón multicapa"*. Paralelizaremos en CPU con **OpenMP** y en GPU con **CUDA**.

Estos modelos se basan en capas compuestas por neuronas, cada neurona aplica una función a los datos de entrada, transformándolos para después enviarlos a las neuronas de la siguiente capa. En nuestro caso tenemos como mínimo 3 capas, la de entrada, la intermedia y la de salida. La capa de entrada son los datos del elemento i-ésimo y no aplican ninguna función. Las neuronas de las capas intermedias (también denominadas ocultas) aplican una función sigmoide[1] sobre los datos que reciben y el resultado lo envían por su salida, que será la entrada de las neuronas de la siguiente capa. La capa de salida tiene tantas neuronas como clases tenga el problema, estas aplican una función softmax[2], que tiene como salida la probabilidad de que el patrón pertenezca a la clase que representa la neurona. Las neuronas se unen entre sí por conectores que pasan la salida de una neurona a otra de la siguiente capa, estas uniones tienen asociado un "peso", que es la importancia de esa neurona en su capa, estos pesos son el objetivo a optimizar de la red, ya que de ellos dependen los resultados.

Un ejemplo de red neuronal es el de la figura 1.1. Sería una red neuronal de tres capas en la cual los patrones de entrada tienen tres atributos, la capa oculta dispone de cuatro neuronas y la de salida de tres, por lo cual este problema tiene tres clases.

En nuestro problema el número de capas ocultas lo determina el usuario, así como el número de neuronas por capa, que será el mismo para todas las capas ocultas. El resto de valores introducidos por el usuario son parámetros que se usan dentro de la red pero no modifican la topología de la red, un ejemplo es la tasa de aprendizaje o el momento.

La creación de la red neuronal se divide en 3 etapas:

- **Inicio**: se crea la red con la topología que el problema y el usuario determinan y con los pesos aleatorios.
- **Entrenamiento**: en el entrenamiento se introducen uno a uno los patrones a la red, por cada patrón introducido se calculan las salidas y se calcula el error (cuanto nos hemos equivocado respecto a lo que deberíamos haber obtenido). Respecto a este error se calcula el ajuste de pesos relativo al patrón y dependiendo si el algoritmo a ejecutar es online u offline se ajustan los pesos por cada patrón o cuando se calcule el ajuste total.
- **Finalización**: se calcula la tasa de acierto de la red (CCR) y se determina si la red es suficientemente buena o no.

Dependiendo de los parámetros introducidos, el CCR puede variar, el CCR nos permite ver la cantidad de patrones bien clasificados, así vemos cómo funciona nuestra red.

En nuestro caso no nos vamos a fijar tanto en que nos dé buenos resultados dependiendo de los parámetros introducidos sino más bien, en la comparativa de tiempos entre serie, **CUDA** y **OpenMP**. Lo que si vamos a tener en cuenta es que el CCR en **serie** y en paralelo sea parecido, así aseguramos que durante la paralelización no se pierde fiabilidad.

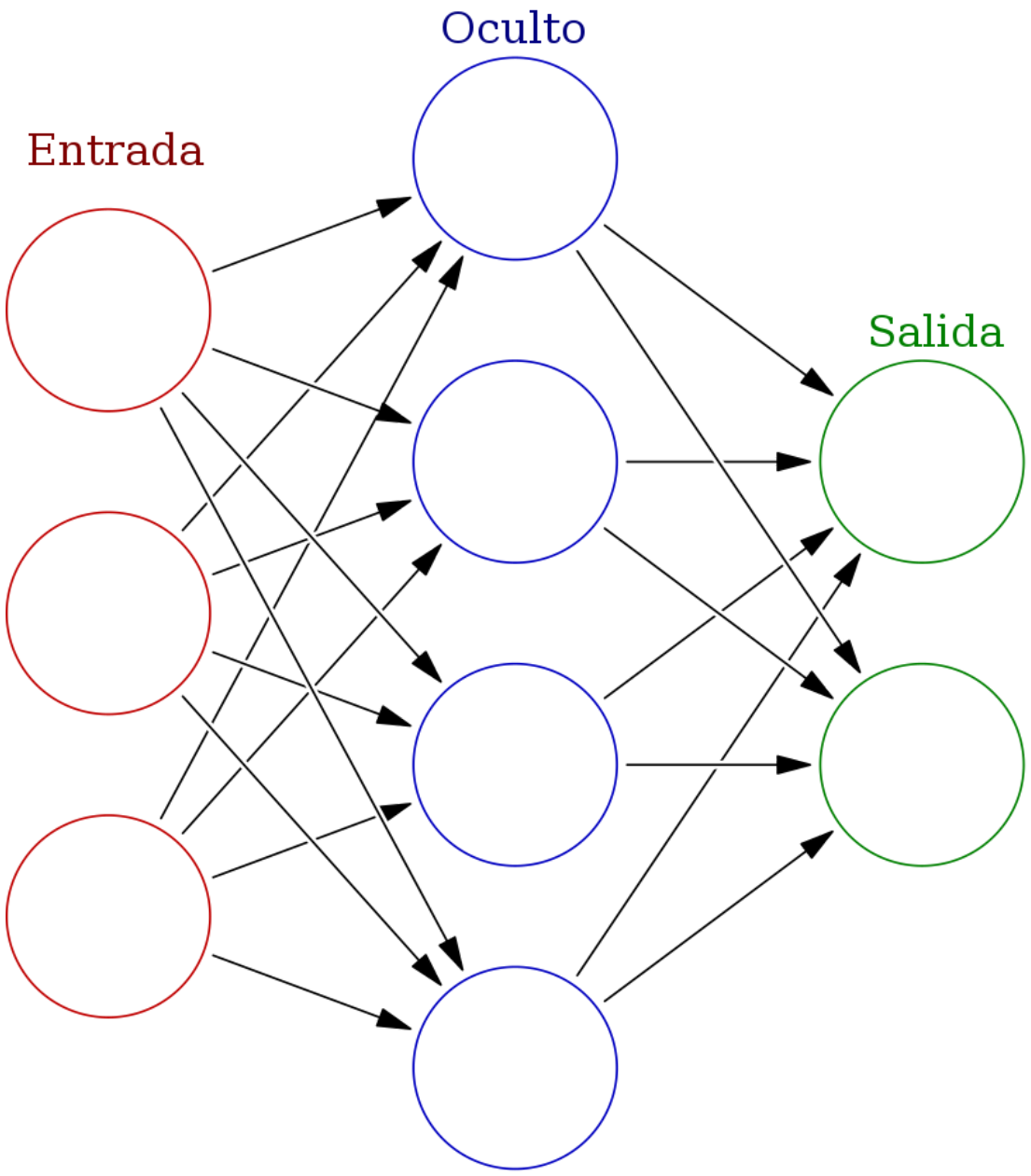


Figura 1.1

Aquí podemos ver el pseudocódigo de los algoritmos online y offline:

Algoritmo de retropropagación *on-line*

Inicio

1. $w_{ji}^h \leftarrow U[-1,1]$ // *Aleatorios entre* -1 *y* $+1$
2. **Repetir**
   - 2.1 **Para** cada patrón con entradas **x**, y salidas **d**
     - 2.1.1 $\Delta w_{ji}^h \leftarrow 0$ // *Se aplicarán cambios por cada patrón*
     - 2.1.2 $out_j^0 \leftarrow x_j$ // *Alimentar entradas*
     - 2.1.3 forwardPropagation() // *Propagar las entradas* $(\Rightarrow\Rightarrow)$
     - 2.1.4 backPropagation() // *Retropropagar el error* $(\Leftarrow\Leftarrow)$
     - 2.1.5 accumulateChange() // *Calcular ajuste de pesos*
     - 2.1.6 weightAdjustment() // *Aplicar el ajuste calculado*

     Fin Para

   Hasta (CondicionParada)
3. **Devolver** matrices de pesos.

Fin

Figura 1.2

Algoritmo de retropropagación *off-line*

Inicio

1. $w_{ji}^h \leftarrow U[-1,1]$ // *Aleatorios entre* -1 *y* $+1$
2. **Repetir**
   - 2.1 $\Delta w_{ji}^h \leftarrow 0$ // *Se aplicarán cambios al final*
   - 2.2 **Para** cada patrón con entradas **x**, y salidas **d**
     - 2.2.1 $out_j^0 \leftarrow x_j$ // *Alimentar entradas*
     - 2.2.2 forwardPropagation() // *Propagar las entradas* $(\Rightarrow\Rightarrow)$
     - 2.2.3 backPropagation() // *Retropropagar el error* $(\Leftarrow\Leftarrow)$
     - 2.2.4 accumulateChange() // *Acumular ajuste de pesos*

     Fin Para
   - 2.3 weightAdjustment() // *Aplicar el ajuste calculado*

   Hasta (CondicionParada)
3. **Devolver** matrices de pesos.

Fin

Figura 1.3

En resumen lo que se busca con este problema es crear una red neuronal que sea capaz de recibir un patrón y decir a qué clase pertenece, para ello, se necesita un entrenamiento previo con muchos patrones para ajustar los pesos, eso es lo que realiza el código a grandes rasgos.

La base de datos que se ha utilizado para comprobar la eficiencia de los distintos códigos ha sido la base de datos **NotMNIST**. La base de datos **NotMNIST** está compuesta por *900* patrones de entrenamiento y *300* patrones de test. Está formada por un conjunto de letras (de la a A la J) escritas con diferentes tipografías o simbologías. Están ajustadas a una rejilla cuadrada de 28×28 píxeles. Las imágenes están en escala de grises en el intervalo [−1,0; +1,0]. Cada uno de los píxeles es una variable de entrada (con un total de 28×28 = 784 variables de entrada) y las clases se corresponden con la letra escrita (a, b, c, d, e, f, g, h, i y j, con un total de 10 clases).

Figura 1.4

2.- Descripción de la arquitectura

La versión secuencial y la paralelizada con **OpenMP** serán codificadas en *C++* y se ejecutarán sobre un procesador tradicional, concretamente sobre un *AMD Ryzen 5 2600*, que consta de 6 cores, cada uno soportando 2 hilos, los que hace un total de 12 hilos de ejecución simultáneos. Cada core consta de una caché L1 de 64Kb, así como una L2 de 512Kb propias, compartiendo dos cachés L3 de 8MB (una por tres cores). El entorno de ejecución será *Linux Mint 20.1 cinnamon*.
Para la versión **CUDA** utilizaremos una tarjeta gráfica *NVIDIA GTX 1060* (versión de 6GB), la cual tiene arquitectura *Pascal*, con 1280 cores repartidos en 20 multiprocesadores de 64 núcleos cada uno. Cada multiprocesador tiene 24Kb de caché L1 y 64Kb de memoria compartida.

Figura 2.1

En la imagen mostrada hay 40 multiprocesadores, en nuestro caso es la mitad, por tanto, la gráfica donde se van a realizar la pruebas sería la mitad de la imagen.

3.- Codificación de problema en serie

3.1.- Codificación del problema

En este apartado nos vamos a centrar en explicar la parte de código que posteriormente se ha paralelizado para que se entienda el procedimiento de serie a paralelo.

Como se explicó anteriormente, el entrenamiento consiste en épocas con cada uno de los patrones:

```
void MultilayerPerceptron::train(Dataset* trainDataset, int errorFunction) {
	if (!online) {
		for (int i = 1; i < nOfLayers; i++) {
			for (int j = 0; j < layers[i].nOfNeurons; j++) {
				for (int k = 0; k < layers[i-1].nOfNeurons+1; k++) {
					if (layers[i].neurons[j].deltaW != NULL) {
						layers[i].neurons[j].deltaW[k] = 0;
					}
				}
			}
		}
	}
	for(int i=0; i<trainDataset->nOfPatterns; i++){
		performEpoch(trainDataset->inputs[i], trainDataset->outputs[i], errorFunction);
	}
	if (!online) {
		weightAdjustment();
	}
}
```

Figura 3.1.1

La época realiza las siguientes operaciones:

```
void MultilayerPerceptron::performEpoch(double* input, double* target, int errorFunction) {
        if (online) {
                for (int i = 1; i < nOfLayers; i++) {
                        for (int j = 0; j < layers[i].nOfNeurons; j++) {
                                for (int k = 0; k < layers[i-1].nOfNeurons+1; k++) {
                                        if (layers[i].neurons[j].deltaW != NULL) {
                                                layers[i].neurons[j].deltaW[k] = 0;
                                        }
                                }
                        }
                }
        }
        feedInputs(input);
        forwardPropagate();
        backpropagateError(target, errorFunction);
        accumulateChange();
        if (online) {
                weightAdjustment();
        }
}
```

Figura 3.1.2

Primero, dependiendo de si el algoritmo va a ser online o offline, es necesario en el caso de que sea online limpiar los *deltaW*, ya que aquí se acumulan los cambios y como en el online se aplican los cambios constantemente por cada época es necesario ponerlos a 0. También si es online es necesario ajustar los pesos como explicamos anteriormente. Una vez aclarado lo anterior se van a explicar todas las funciones que constituyen una época:

feedInputs: En esta función se alimentan todas las neuronas de entrada con todos los valores de entrada del patrón.

forwardProgapate: En esta función se aplica la propagación hacia delante, esto consiste en que, cada neurona realiza la suma de todas las neuronas de la capa anterior por los pesos del enlace, una vez calculada la suma se aplica la función de activación que para nuestras redes son sigmoides, excepto en la última capa que son *SoftMax*.

backpropagateError: una vez se realiza la propagación adelante se calcula el resultado y se ve cuanto a fallado respecto al resultado esperado que nos da el patrón, en nuestro caso se va a calcular con la función de entropía cruzada.
Este error obtenido se propaga hacia atrás aplicando una serie de fórmulas y derivadas permitiendo calcular unos nuevos pesos que permiten reducir ese error.

acumulateChange: los pesos obtenidos los almacenamos en *deltaW* donde vamos a ir acumulando todos los cambios en los pesos.

weightAdjustmentOnline: esta función aplica los pesos, es decir, los cambios que teníamos en *deltaW* que se iban acumulando (solo en el caso de offline), se aplican a los pesos de cada neurona aplicando una fórmula donde se indica la tasa de aprendizaje y el momento para especificar cuánto aprende la red, es decir, cuánto afectan los patrones y sus respectivos pesos obtenidos respecto a los pesos anteriores.

3.2.- Resultados obtenidos del problema en serie

Haremos varias pruebas con distintas topologías de la red (modificando tanto el número de capas entre 1, 2 y 4 como el número de neuronas por capa siendo 16, 64 y 32) y con 2 y 5 entrenamientos. Los resultados obtenidos son los siguientes:

- Para 1 capa y 16 neuronas:
  - 2 entrenamientos: 60.087
  - 5 entrenamientos: 151.271

- Para 2 capas y 64 neuronas:
  - 2 entrenamientos: 261.724
  - 5 entrenamientos: 649.266

- Para 4 capas y 32 neuronas:
  - 2 entrenamientos: 142.672
  - 5 entrenamientos: 351.361

Como podemos observar, cuantos más enlaces tiene la red neuronal, más tiempo tarda en ejecutarse el programa, esto es lógico, ya que a más enlaces, mayor cantidad de cálculos van a ser necesarios realizar (por ejemplo el cálculo del nuevo peso tras realizar las derivadas para cada neurona o incluso la propagación de las entradas hacia delante pasando por todas las neuronas en la red neuronal).

4.- Paralelización con OpenMP

4.1.- Adaptación del código en OpenMP

Para realizar la paralelización con **OpenMP**, nos hemos apoyado en las directivas que **OpenMP** nos ofrece, las que nosotros principalmente hemos utilizado son las siguientes:

- **omp_set_num_threads()**, esta función nos permite especificar el número de hilos que se van a aplicar a la paralelización.
- **pragma omp parallel for [cláusulas]**, esta función permite especificar que el siguiente bucle for se va a paralelizar (en el caso de haber especificado los hilos, se usarán los hilos especificados para el bucle for), la cláusula que nosotros usamos es private, con esta cláusula hacemos que la variable especificada sea privada para cada hilo y así evitar que varios hilos modifiquen esa variable.

Las paralelizaciones que hemos realizado las podemos clasificar en tres tipos:

-**Paralelizaciones pequeñas o de menor importancia:** Las paralelizaciones que consideramos que pertenecen a este grupo son aquellas que se van a notar en menor medida como la paralelización de reserva de memoria o incluso la liberación de memoria, esto consiste en paralelizar la reserva de espacio para cada neurona permitiendo agilizar un poco este proceso (mientras mayor sean las redes más se notará esta paralelización). Un ejemplo de este tipo de paralelización se puede apreciar en la siguiente imagen donde hemos realizado paralelización al liberar memoria:

```
#pragma omp parallel for private(i)
for (i = 1; i < nOfLayers; i++) {
        for (int j = 0; j < layers[i].nOfNeurons; j++) {
                if (layers[i].neurons[j].w != NULL) {
                        delete[] layers[i].neurons[j].w;
                        delete[] layers[i].neurons[j].wCopy;
                        delete[] layers[i].neurons[j].deltaW;
                        delete[] layers[i].neurons[j].lastDeltaW;
                }
        }
        delete[] layers[i].neurons;
}
delete[] layers;
```

Figura 4.1.1

Tiempo con solo paralelizaciones menores: 259.149s.

-**Paralelización de todo el programa:** El programa en serie está diseñado para hacer 5 entrenamientos y así obtener un resultado medio de ellos. Para paralelizar esta parte decidimos que en vez de 5 entrenamientos se hiciesen *n* (ntest), y cada entrenamiento se asocia a un hilo, por lo cual, si decidimos hacer 5 entrenamientos tendremos 5 hilos en el bucle externo:

```
int i;
omp_set_num_threads(ntest);
#pragma omp parallel for private(i)
            for( i=0; i<ntest; i++){
                    /*cout << "**********" << endl;
                    cout << "SEED " << seeds[i] << endl;
                    cout << "**********" << endl;*/
                    srand(seeds[i]);
                    mlp[i].runBackPropagation(trainDataset,testDataset,maxIter,&(trainErrors[i]),
    &(testErrors[i]),&(trainCCRs[i]),&(testCCRs[i]),&(count[i]), i);
    std::cout << "semilla["<<i<<"]" << '\n'
                     << "We end!! => Final test CCR: " << testCCRs[i] << '\n'
     << " We end!! => Final train CCR: " << trainCCRs[i] << endl;
            }
```

Figura 4.1.2

Tiempo paralelizando solo bucle externo: 133.202s.

-**Paralelización del algoritmo:** nuestro objetivo principal consistía en buscar algún método para mejorar la velocidad del algoritmo buscando donde se podía paralelizar y si era rentable.

Llegamos a la conclusión de que se podían paralelizar las épocas (concepto explicado anteriormente), es decir, creamos una cantidad de hilos N y a cada hilo le asignamos X patrones a entrenar del total de patrones, por ejemplo, si tenemos 120 patrones y creamos 4 hilos, cada hilo tendrá que realizar las épocas para 30 patrones. Este entrenamiento es offline, lo que queremos decir es que cada hilo realiza el entrenamiento con sus respectivos patrones y estos dan sus respectivos cambios acumulados, cada vez que se hace un patrón no se ajustan los pesos, sino que se acumulan todos. Cuando todos han acumulado sus pesos, se suman las acumulaciones de los distintos hilos y posteriormente se ajustan los pesos.

Para poder realizar lo anterior, es necesario crear redes copia, es decir, para nuestra red neuronal se necesitan tantos clones de esa red neuronal como hilos tenemos, reservando la memoria para cada red neuronal de copia. Esto genera que el tamaño del programa aumente pero nosotros buscamos mejorar la eficiencia del algoritmo.

Como podemos observar al entrenar 5 redes a la vez de forma paralela y dentro de cada hilo creamos sub-hilos para la ejecución de las épocas, por lo tanto debemos tener cuidado

a la hora de asignar el número de hilos que las ejecutan para no crear más hilos de los que disponga nuestro dispositivo, ya que así empezarían a competir por los recursos.
En la siguiente imagen podemos observar la implementación de la división de los patrones por cada hilo creado, la creación de los hilos y la suma de los resultados de los mismos:

```
int j, i;
int patronesPorHilo = trainDataset->nOfPatterns/nThreads;
copyOriginalToCopys();
omp_set_num_threads(nThreads);
#pragma omp parallel for private(i,j)
for(i=0; i<nThreads; i++){
        for (j = i*patronesPorHilo; j < (i*patronesPorHilo)+patronesPorHilo; j++) {
                performEpoch(trainDataset->inputs[j], trainDataset->outputs[j], i);
        }
}
sumNetworks();

//paralelizar si offline
/*for(int i=0; i<trainDataset->nOfPatterns; i++){
        performEpoch(trainDataset->inputs[i], trainDataset->outputs[i], sdda);
}*/
if (!online) {
        weightAdjustment();
}
```

Figura 4.1.3

Tiempo paralelizando solo bucle interno: 201.449s.

4.2.- Resultados experimentales obtenidos con openmp

Empezaremos por comparar varios casos de paralelización, comenzaremos paralelizando solo el bucle externo, el bucle interno, y las paralelizaciones menores, después se realizarán pruebas combinando ambas y finalmente usando todas las paralelizaciones a la vez. Estas pruebas se realizarán con una topología de dos capas ocultas y 64 neuronas por capa, con dos entrenamientos (iteraciones del bucle externo) y 2 hilos para el bucle interno.

Tiempo serie: 261.724s.
Tiempo con solo paralelizaciones menores: 259.149s.
Tiempo paralelizando solo bucle externo: 133.202s.
Tiempo paralelizando solo bucle interno: 201.449s.
Tiempo paralelizando bucle interno y paralelizaciones menores: 200.422s.
Tiempo paralelizando bucle externo y paralelizaciones menores: 134.162s.
Tiempo paralelizando bucle externo e interno, sin paralelizaciones menores: 132.945s.
Tiempo con paralelización completa: 122.498s.

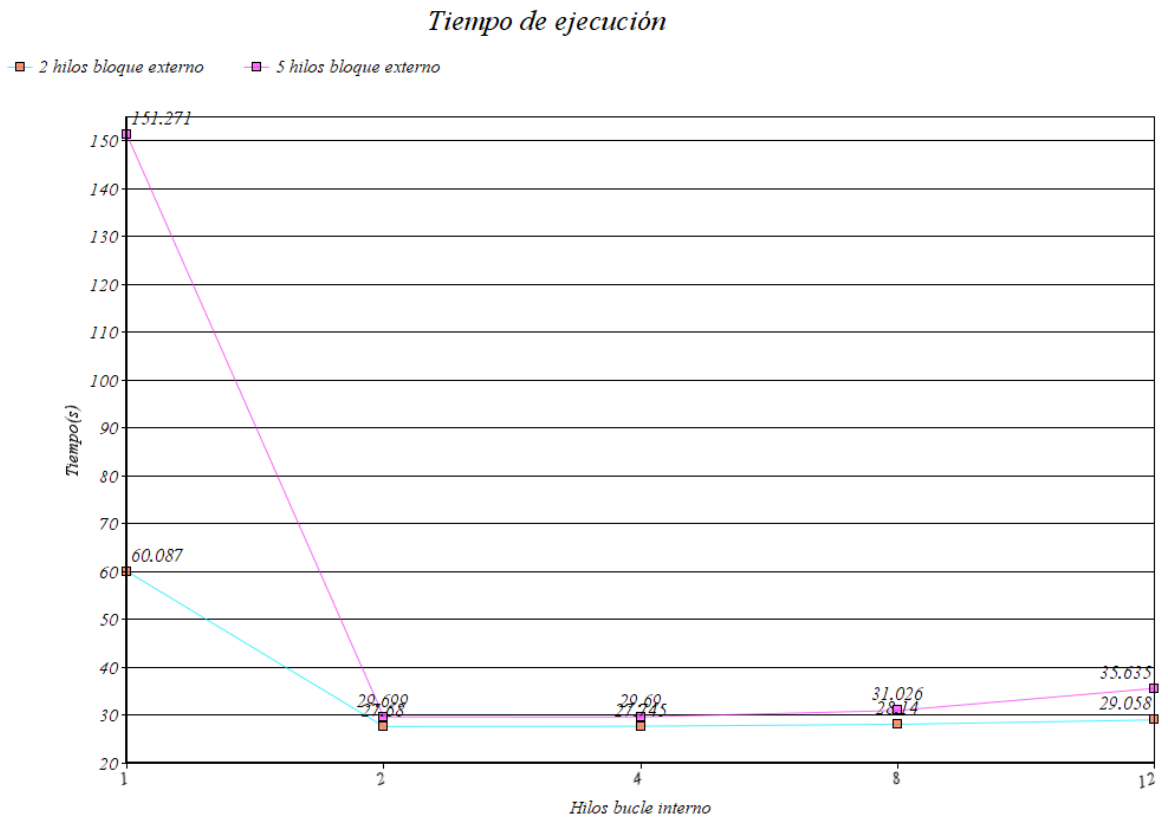
Tabla con los tiempos obtenidos de todas las pruebas:

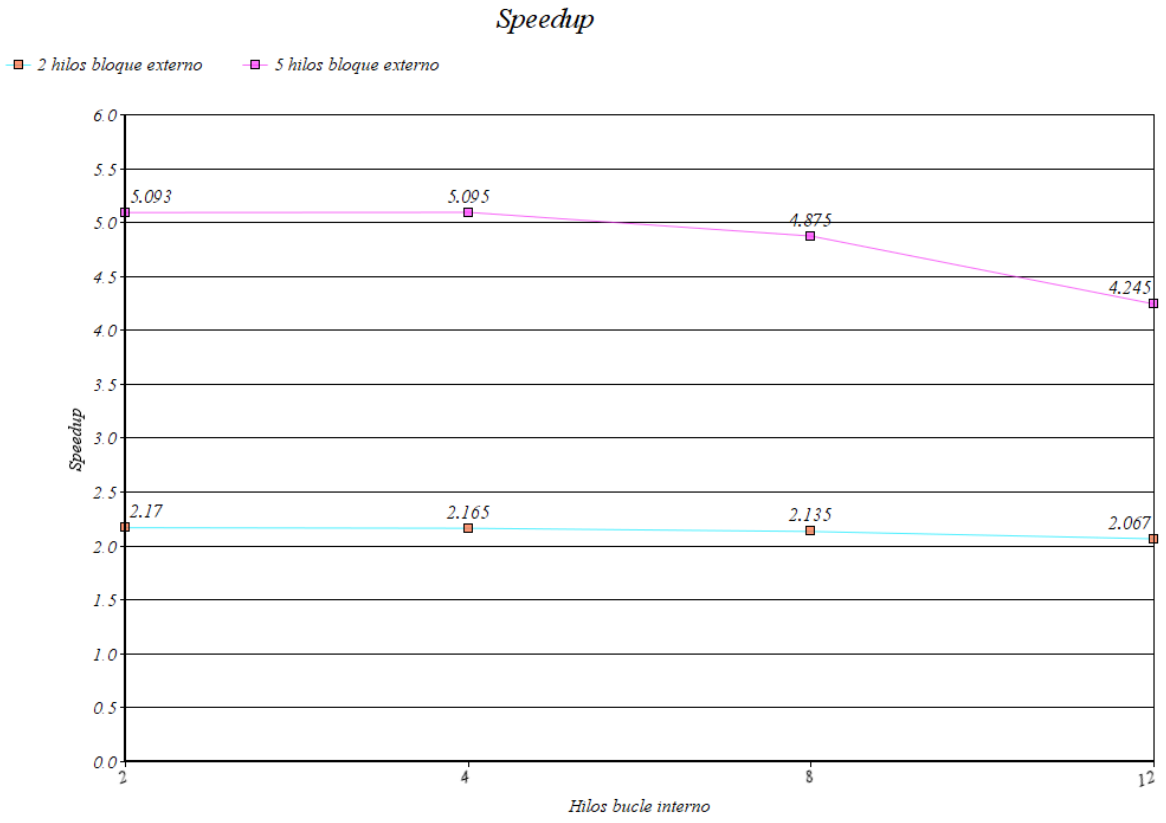
Paralelizaciones	**Bucle Externo**	**Bucle Interno**	**Menores**
Bucle Externo	133.202s	-	-
Bucle Interno	132.945s	201.449s	-
Menores	134.162s	200.422s	259.149s

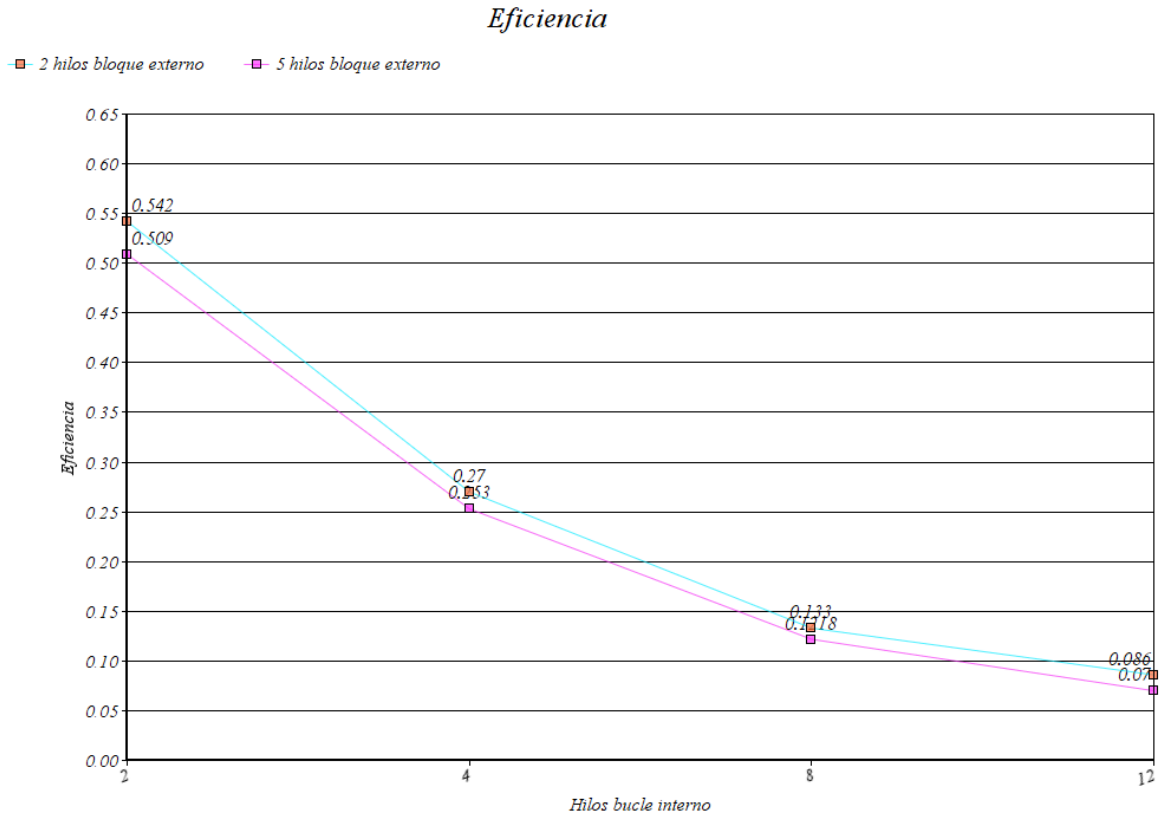
Tiempo para el programa en **Serie**: 261.724s.
Tiempo para el programa con **todas las paralelizaciones juntas**: 122.498s.

A continuación presentaremos los resultados de la paralelización de los casos en serie con distinto número de hilos. Las gráficas las separaremos por *tiempo de ejecución*, *speedup* y *eficiencia*. A su vez distinguiremos entre las tres topologías propuestas para la red. Para la eficiencia multiplicamos los hilos del bucle interno por los hilos del bucle externo para obtener el número de hilos totales usados. Los resultados obtenidos son los siguientes:

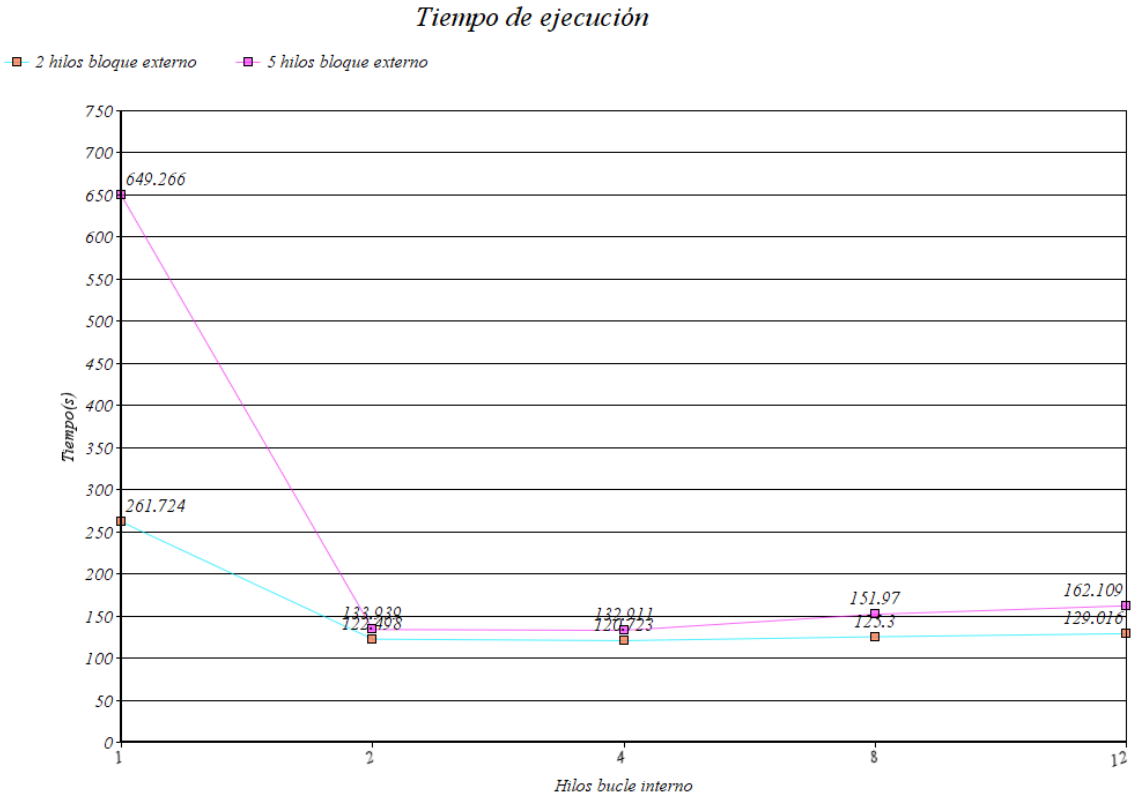
- Experimento con 1 capa oculta y 16 neuronas por capa:

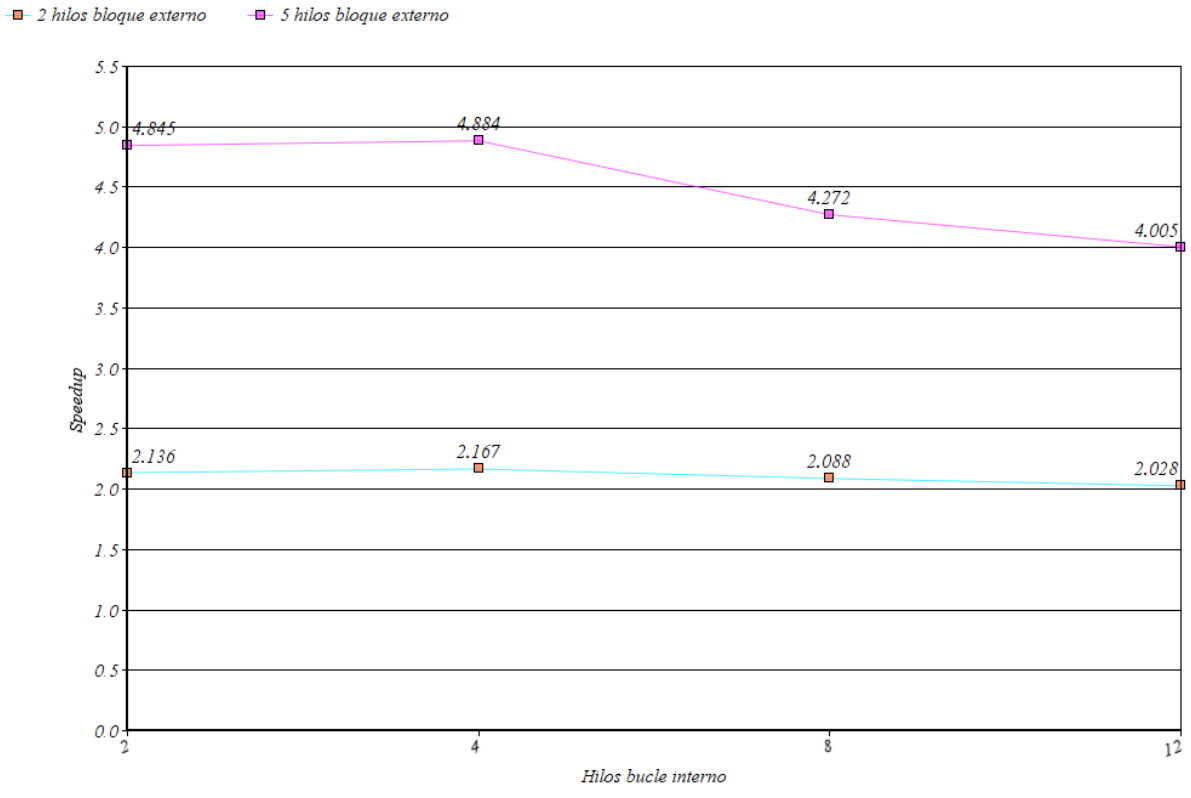


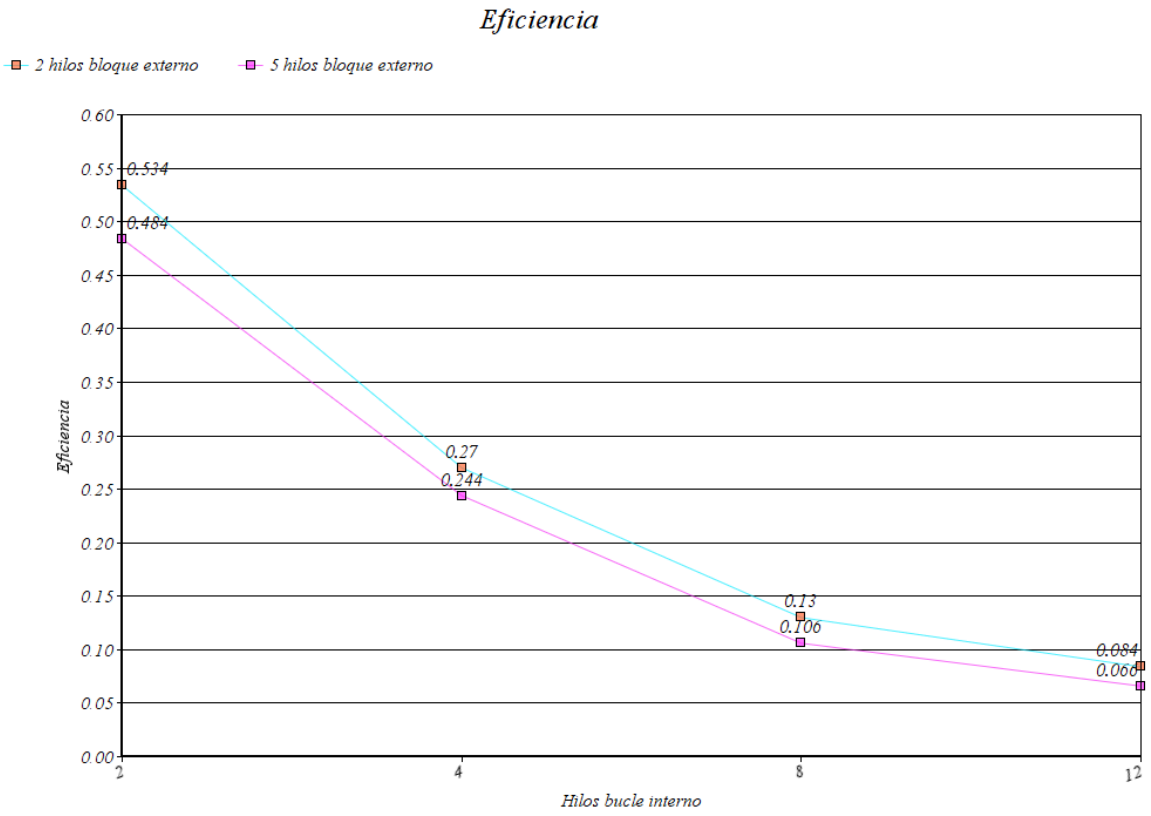

Speedup
2 hilos bloque externo
5 hilos bloque externo
6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Speedup
5.093
5.095
4.875
4.245
2.17
2.165
2.135
2.067
2
4
8
12
Hilos bucle interno


Eficiencia
2 hilos bloque externo
5 hilos bloque externo
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
Eficiencia
0.542
0.509
0.27
0.253
0.133
0.118
0.086
0.07
2
4
8
12
Hilos bucle interno

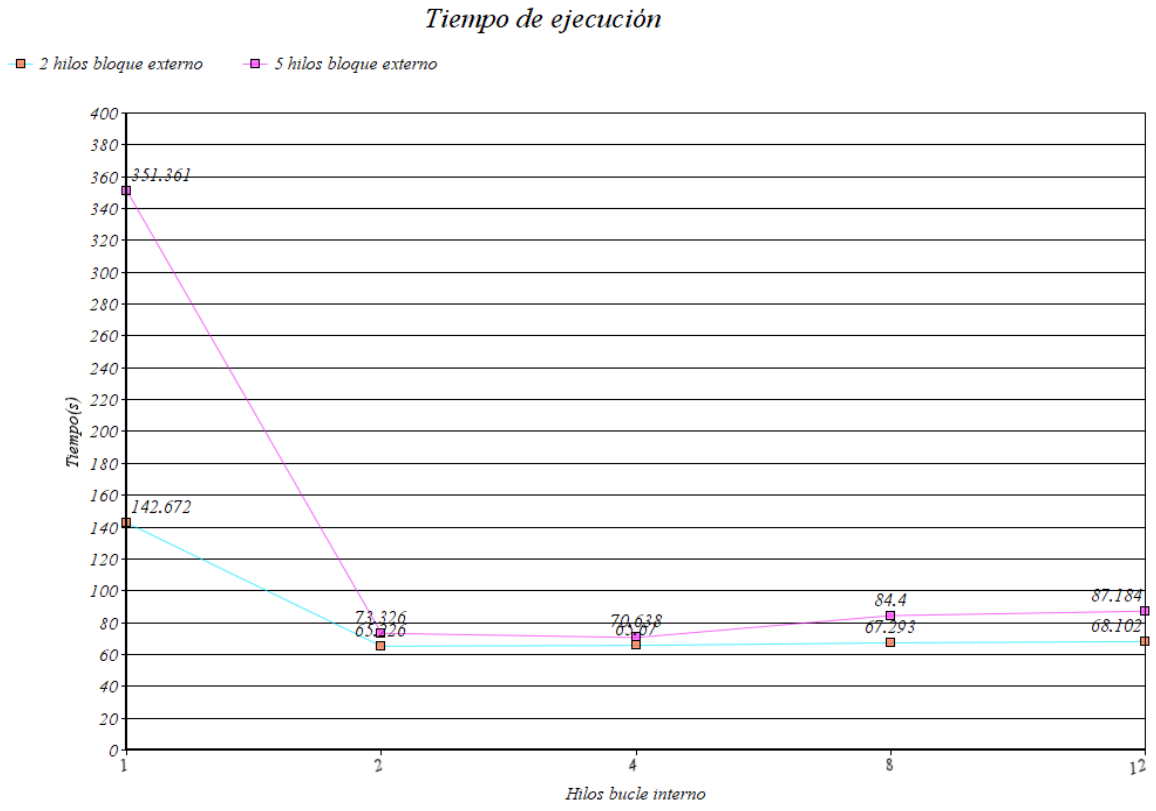
- Experimento con 2 capas ocultas y 64 neuronas por capa:

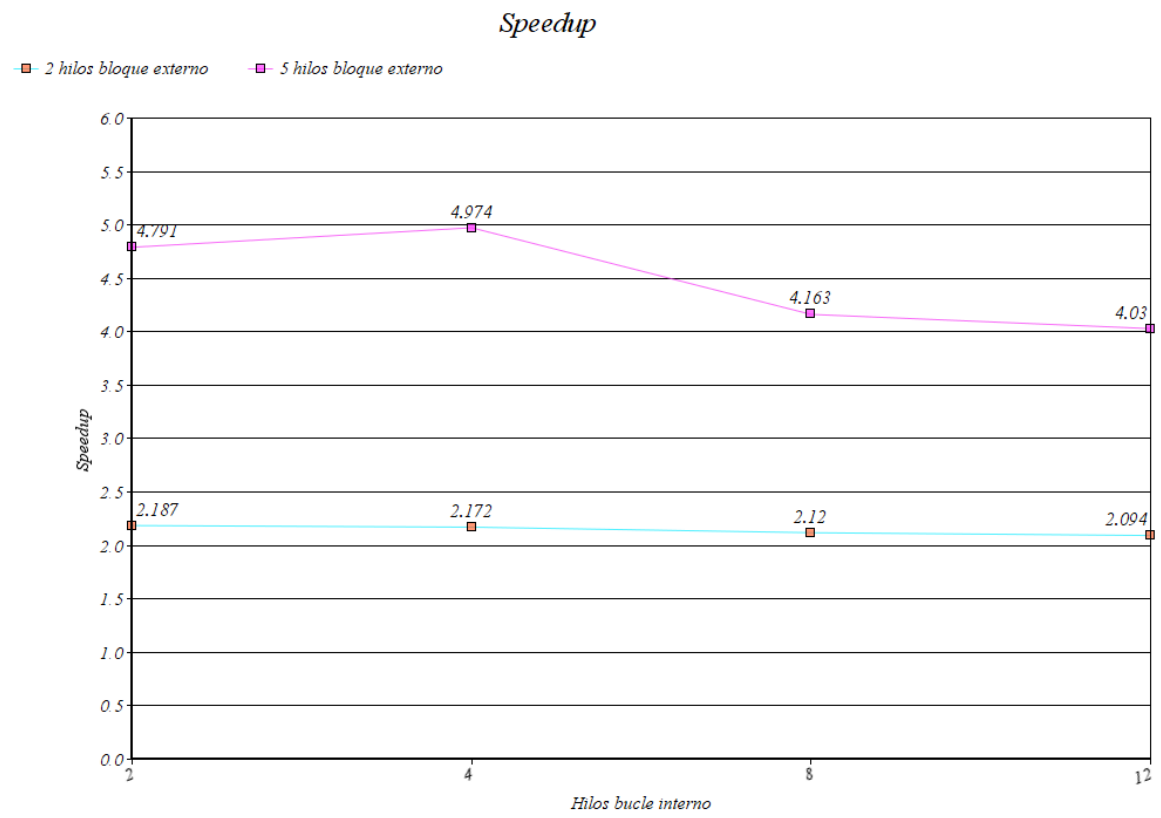


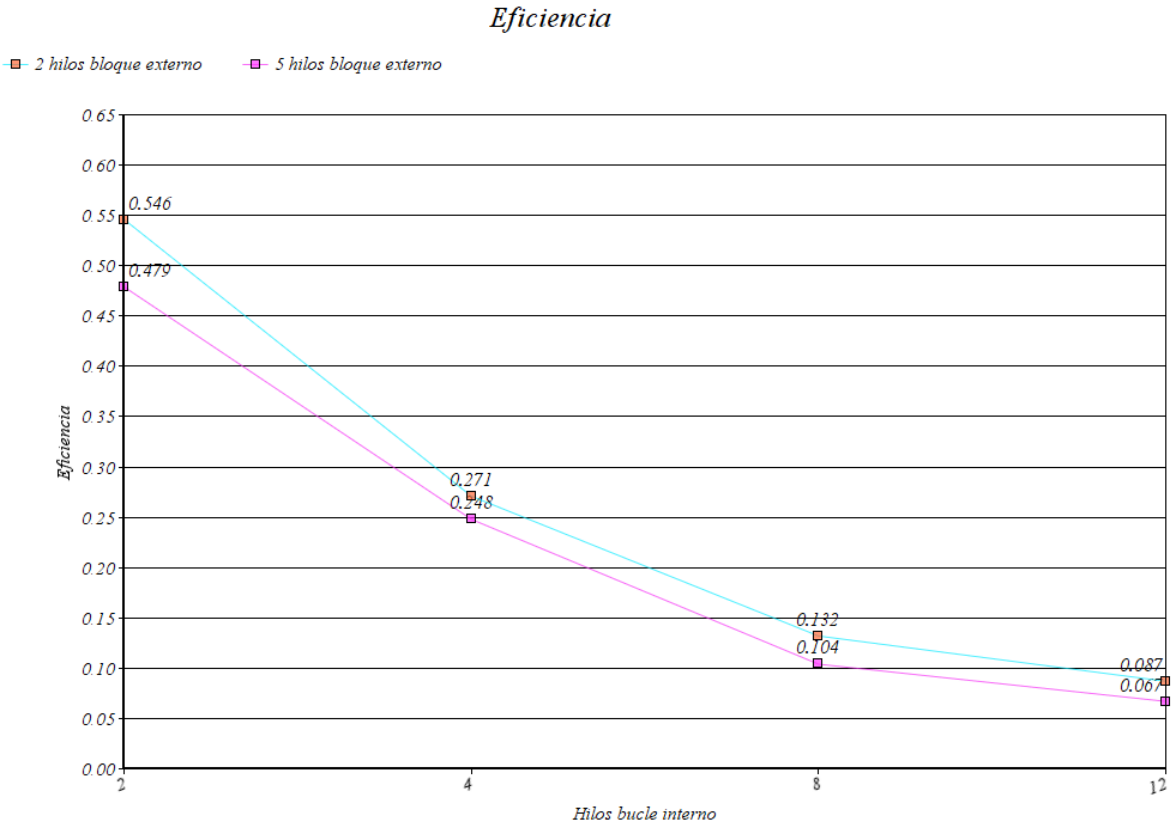




- Experimento con 4 capas y 32 neuronas por capa:




Speedup
2 hilos bloque externo
5 hilos bloque externo
6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Speedup
4.791
4.974
4.163
4.03
2.187
2.172
2.12
2.094
2
4
8
12
Hilos bucle interno


Eficiencia
2 hilos bloque externo
5 hilos bloque externo
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
Eficiencia
0.546
0.479
0.271
0.248
0.132
0.104
0.087
0.067
2
4
8
12
Hilos bucle interno

Tenemos que destacar que las pruebas realizadas con 2 entrenamientos y las pruebas realizadas con 5 entrenamientos no son comparables ya que el número de entrenamientos no es el mismo. Lo que sí podemos ver, es que, el *speedup* es mucho mayor para el caso de 5 entrenamientos, esto es normal ya que tenemos mucho más trabajo que hacer en serie, sin embargo, sigue siendo el mismo trabajo por hilo, ya que cada entrenamiento está asignado a un hilo.

Observamos como prácticamente siempre el mejor caso en relación *speedup/eficiencia* es usar 2 hilos internos. Esto se explica debido a que al aumentar el número de hilos también aumentamos el trabajo de sincronización de memoria, por lo cual es menos eficiente y puede que hasta menos rápido que usar simplemente menos hilos.

También se observa como para el caso de 5 hilos externos tenemos menos capacidad para poner hilos internos ya que al constar de un procesador con 12 cores, si metemos más de 2 hilos internos, superamos este número y los hilos empiezan a competir por los recursos.

Cabe destacar que los tiempos obtenidos realizando paralelización son superiores a los tiempos en serie, es lógico, ya que repartimos la cantidad de trabajo entre varios hilos aumentando la velocidad de cómputo (por ejemplo realizar entrenamientos en paralelo o la mejora realizada al algoritmo explicada anteriormente).

5.- Paralelización en CUDA

5.1.- Adaptación del problema en CUDA

Para entender cómo hemos paralelizado en cuda, es necesario explicar algunos conceptos básicos que usaremos después en la explicación, los conceptos son los siguientes:

- **Host**. Cuando hablamos de host nos referimos a la CPU de nuestro dispositivo.
- **Device**. Cuando nos referimos a device, nos referimos a la GPU de nuestro dispositivo.
- **Capacidad de cómputo**. Con este concepto nos referimos a la versión que determina las características CUDA disponible, en este caso nos referimos a la arquitectura para entender las capacidades de la GPU.
- **Kernel**. Es la parte del programa que se encuentra listo para ejecutar en la GPU.
- **Warp**. Es la unidad mínima que recomienda NVIDIA para trabajar con hilos, en este caso sería un Warp serían 32 hilos.
- **Bloque**. Es el conjunto de hilos que comparten memoria.
- **Grid**. Es el conjunto de bloques que conforman el Kernel
- **Core**. Se encarga de ejecutar los hilos.
- **Multiprocesador**. Es un conjunto de cores donde se ejecutan los bloques.

Una vez explicado esto, para realizar la paralelización con CUDA, nos hemos apoyado en las directivas que **CUDA** nos ofrece, las que nosotros principalmente hemos utilizado son las siguientes: cudaMalloc, cudaMemcpy, cudaFree, cudaDeviceSynchronize, función kernel.

cudaMalloc: esta función nos permite reservar memoria en la Gpu, la estructura de esta función es la siguiente:

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

cudaMemcpy: esta función nos permite copiar la memoria del host al device, la estructura de esta función es la siguiente:

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

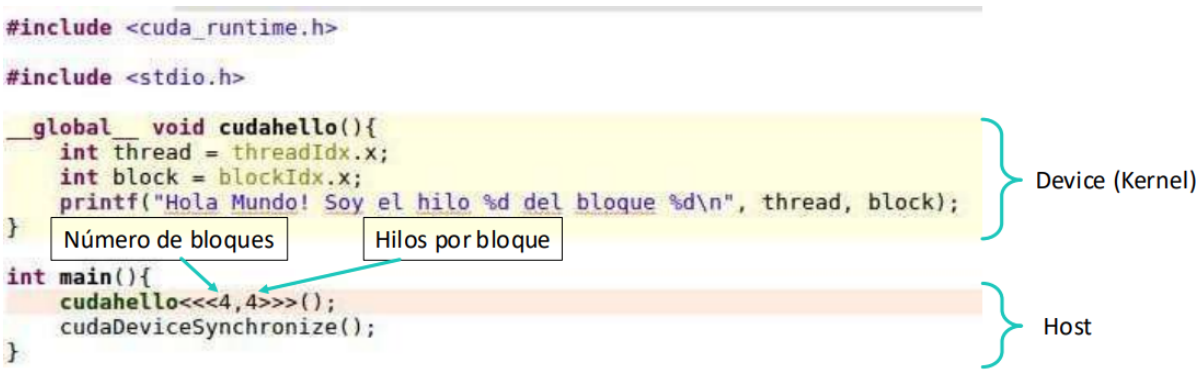
cudaFree: esta función nos permite liberar la memoria de las reservas de memoria creadas en la Gpu, la estructura de esta función es la siguiente:

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

cudaDeviceSynchronize: esta función espera que el cómputo realizado por la Gpu finalice, la estructura de esta función es la siguiente:

```
cudaDeviceSynchronize(void);
```

<u>función Kernel:</u> en esta función se especifica el código que se va a ejecutar en la Gpu, para llamar esta función es necesario especificar el número de bloques y los hilos por bloque que va a usar la Gpu, un breve ejemplo sería el siguiente:



Una vez explicado todo lo anterior ya podemos empezar a hablar de nuestra implementación. Para empezar, tenemos que conocer la capacidad de cómputo de nuestra GPU (necesitamos saber la arquitectura que tenemos, la explicamos en el apartado 2.- Descripción de la arquitectura). Cuda nos ofrece una serie de directivas para saber la capacidad de cómputo de nuestra gráfica, nosotros hemos realizado el siguiente ejemplo para averiguarlo:

```
1
2   #include <cuda_runtime.h>
3   #include <stdio.h>
4
5   int main()
6   {
7       int deviceCount;
8       cudaGetDeviceCount(&deviceCount);
9       for (int device = 0; device < deviceCount; device++) {
10          cudaDeviceProp deviceProp;
11          cudaGetDeviceProperties(&deviceProp, device);
12          printf("Device %d (%s) has compute capability %d.%d. \n",
13          device, deviceProp.name, deviceProp.major, deviceProp.minor);
14      }
15
16      return 0;
17  }
```

En este código primero se obtiene el número de device disponibles que tenemos con **CudaGetDeviceCount**, con esta cantidad (en nuestro caso uno), podemos recorrer todos esos dispositivos y obtener sus propiedades con **CudaGetDeviceProperties**.

Una vez tenemos la propiedades del dispositivo, en el parámetro *.major* del objeto que habíamos creado anteriormente para guardar la propiedades, obtenemos la capacidad máxima.

Dependiendo de este valor podremos saber de qué tipo de arquitectura estamos hablando, de tal forma que si **major=deviceProp.major** y **minor=deviceProp.minor**, entonces:

major=1 → Arquitectura **Tesla** (8 cores)
major=2 → Arquitectura **Fermi** (si minor es 0 son 32 cores si no son 48)
major=3 → Arquitectura **Kepler** (192 cores)
major=5 → Arquitectura **Maxwell** (128 cores).
major=6 → Arquitectura **Pascal** (64 cores, este es nuestro caso)
major=7 → Arquitectura **Volta** si minor es 0 (64 cores) o Arquitectura **Turing** si minor es distinta de 0.
major=8 → Arquitectura **Ampere** (64 cores)
Si major no tiene ninguno de estos valores → la arquitectura es desconocida.

Una vez realizado este paso, intentamos realizar una reserva de memoria de nuestras estructuras a la Gpu para su posterior copia, llamada al kernel, sincronización y liberación de memoria (como hemos visto en los comandos explicados anteriormente), pero nos encontramos con una serie de problemas.

5.2.- Problemas encontrados al implementar Cuda en nuestro código.

Al empezar a programar en **CUDA** nos encontramos con varios errores al intentar enviar punteros de nuestras estructuras a nuestro *dispositivo* (tarjeta gráfica). El problema con el que nosotros nos hemos topado es que, mientras que pasar un puntero a una estructura definida por el sistema como puede ser un entero o un flotante es una tarea trivial, pasar una estructura es un trabajo más complejo. Esto se debe a que el *host* y el *dispositivo* tienen distintos espacios de memoria, y cada uno tiene un direccionamiento distinto, por lo cual, cuando hacemos un *cudaMalloc* reservamos memoria en el *dispositivo* para más tarde, con *cudaMemcpy,* copiar la memoria del *host* al espacio reservado y direccionado en el dispositivo.
El problema con las estructuras que tienen punteros dentro es que, si intentamos copiar la estructura directamente con *cudaMemcpy*, nos dará error, ya que el *dispositivo* no tiene direccionados los punteros de dentro de esa estructura, por lo cual el direccionamiento se complica.
La forma de pasar una estructura no es intuitiva, aunque tiene sentido si entendemos cómo funciona la memoria. Estos dos puntos se explicarán a continuación de una forma más detallada y con un ejemplo creado por nosotros, donde se consiguió pasar una estructura, modificarla en la *función kernel* y copiar la estructura modifica en el *host,* en un problema más sencillo y más entendible.

5.3.- Solución a los problemas encontrados.

Tras no ser capaces de pasar la estructura del *host* al *device* y poder modificarla en el *device* para que posteriormente se pudiera enviar de vuelta del *device* al *host*, decidimos realizar un ejemplo a menor escala para entender mejor el paso de las estructuras.
El ejemplo que nosotros realizamos y que conseguimos que funcionara fue el siguiente:

```
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;
```

Primeramente creamos una estructura sencilla llamada *Matrix* que consta de dos valores enteros que son el alto y el ancho, y un vector float donde se almacenará la matriz en sí.

```
Matrix* C = NULL;
C = new Matrix;
C->height = 5;
C->width = 5;
C->elements = new float[C->height * C->width](); // initialize it all to '0'
```

Una vez tenemos la estructura definida, creamos en primer lugar la matriz C, como vemos en la imagen anterior, se inicializan las variables y se realiza la reserva de la memoria tanto de la estructura *Matrix* como del vector float (en realidad es una matriz si nos fijamos, ya que reservamos memoria tanto para el alto como el ancho, es una matriz en un solo vector).

```
Matrix* hostMatrix = C; // let
Matrix* deviceMatrix = NULL;
float*  d_elements;
```

Una vez tenemos la matriz C creada, es necesario crear una copia en el host para pasar esa copia al device, en la imagen anterior se pueden ver la creación de punteros necesarios para poder realizar el paso de *host* a *device* y de *device* a *host*. *hostMatrix* es un puntero a nuestra estructura que se encuentra en el *host*, *deviceMatrix* será el puntero que se encuentre en el *device* y *d_elements* es un puntero donde se encontrará el vector float que tenemos definido en la estructura para el *device* ya que es las direcciones de memoria son diferentes entre el *host* y el *device* por eso en la propia estructura es necesario especificar dónde se va a realizar el espacio de memorias en el *device* y donde se va a almacenar, se podrá comprender mejor viendo las siguientes imágenes.

```
// allocate the deviceMatrix and d_elements
cudaMalloc(&deviceMatrix, sizeof(Matrix));
int size = hostMatrix->width * hostMatrix->height * sizeof(float);
cudaMalloc(&d_elements, size);
```

En la imagen anterior usamos *cudaMalloc* para realizar la reserva de memoria en el *device* de la estructura, especificando su tamaño. También es necesario reservar memoria a la matriz que se encuentra dentro de la estructura, esto lo conseguimos reservando memoria a *d_elements*, que es el puntero que explicamos anteriormente, donde especificamos el tamaño que va a ocupar en memoria.

```
// copy each piece of data separately
cudaMemcpy(deviceMatrix, hostMatrix, sizeof(Matrix), cudaMemcpyHostToDevice);
cudaMemcpy(d_elements, hostMatrix->elements, size, cudaMemcpyHostToDevice);
cudaMemcpy(&(deviceMatrix->elements), &d_elements, sizeof(float*), cudaMemcpyHostToDevice);
```

Una vez reservamos esa memoria se realizan las copias del *host* a *device*, para ello, primero copiamos la estructura del *host* a *device*, esto lo conseguimos pasando el puntero host al puntero device donde su tamaño debe coincidir por las especificaciones anteriores y le decimos aparte a *cudaMemcpy* el tamaño de la copia que es del tamaño de la estructura *Matrix* (*cudaMemcpyHostToDevice* especifica que la copia va a ser de *host* a *device* y no al contrario).
Una vez realizado el paso anterior, realizamos la copia de los elementos de la matriz de dentro de la estructura *Matrix* (de C que es la del *host*), por ello, pasamos los elementos que se encuentran en el puntero *hostMatrix->elelments* a *d_elements*, los tamaños reservados para los dos punteros anteriores deben de coincidir ya que a ambos le asignamos un tamaño reservado de size que viene a ser columnas por filas (de tipo float).
Cuando tenemos los datos copiados en *d_elementos* es necesario especificar que esos elementos pertenecen a la estructura del *device*, entonces, por último, copiamos el puntero que es *d_elements* a la variable de la estructura *deviceMatrix* que es elements y así tendríamos especificado el puntero que se encuentra dentro de la estructura en el *device*.

```
hello << <1, 1 >> > (deviceMatrix);
```

Cuando la reserva de la memoria y el paso de la información de *host* a *device* ha finalizado, llamamos a la *función kernel* para comprobar si se pasa correctamente y puede modificar valores de la estructura, por tanto le pasamos la estructura creada para el *device* que es *deviceMatrix*, los valores puestos 1 y 1 hacen referencia al número de bloque e hilos que se van a utilizar como es una prueba y no vamos a realizar paralelización en este caso, pasamos un único bloque y un único hilo, quiero recordar que para los hilos como recomienda **CUDA**, siempre es necesario usar **Warp**, es decir, usar 32 hilos cuando se quiera paralelizar para utilizar mejor la memoria que se le pasa al *device*.

```
__global__ void hello(Matrix* deviceMatrix) {
    for(int i=0; i<deviceMatrix->width*deviceMatrix->height;i++){
        deviceMatrix->elements[i] = 5;
    }
}
```

En la imagen anterior podemos ver la *función kernel* (el código que se ejecuta en el *device*), en nuestro caso, únicamente añadimos cincos a la matriz creada dentro de la estructura que pasamos desde el *host* al *device* para comprobar si funciona correctamente, no se realiza paralelización.

```
cudaMemcpy(hostMatrix->elements, d_elements, size, cudaMemcpyDeviceToHost);
```

Por último, es necesario volver a copiar la memoria del *device* al *host* y esto lo conseguimos copiando los elementos que han sido modificados, como son los elementos que se encuentran dentro del puntero *d_elements* (elementos que se encuentran dentro de la estructura del *device*) y almacenando esos elementos de nuevo en los elementos de nuestra matriz de la estructura que se creó en el *host* (*hostMatrix* en nuestro caso, en la variable *elements* que es un puntero float).

Una vez realizados todos estos pasos, y al ejecutar el programa, comprobamos que el ejemplo funciona de manera correcta pero nos damos cuenta de lo complejo que puede llegar a ser para un caso sencillo como puede ser este.

5.4.- Conclusiones obtenidas respecto a la implementación en CUDA.

Tras conocer los problemas que nos hemos encontrado a la hora de intentar paralelizar usando **CUDA** en nuestro código (se comenta en el apartado anterior), hemos llegado a las siguientes conclusiones:

- **El uso de estructuras a la hora de paralelizar con CUDA no es eficiente si nos basamos en el uso de memoria**, como vimos en el apartado anterior, si la estructura tiene punteros internos es necesario especificarlos y reservarlos dentro del dispositivo, de esta forma, esta memoria reservada no tiene por que encontrarse junto a la estructura, por tanto, no reservamos la memoria en bloques sino que pueden haber saltos en la memoria produciendo mala eficiencia la recoger datos o repartir al repartir entre los hilos.

- **El uso de estructuras a la hora de paralelizar con CUDA no es eficiente si nos basamos en la implementación**, la implementación del paso de estructuras con punteros de *host* a *device* es poco intuitivo y complejo de implementar (una vez lo consigues realizar se comprenden las dificultades pero sigue siendo un proceso tedioso), por tanto, a la hora de paralelizar se perdería demasiado tiempo en conseguir pasar únicamente todas las estructuras al *device*.

Como describimos en los puntos anteriores, nuestro problema se basa principalmente en el uso de estructuras y no de matrices, si fueran matrices es paso de la información sería mucho más sencillo y aparte, sería mucho más eficiente (se controlaría mejor la memoria y es uso de la misma para cada hilo).

Por tanto, para nuestro caso, no vale la pena ni computacionalmente ni en inversión de tiempo paralelizar el *algoritmo de Perceptrón multicapa* con **CUDA**, por los motivos expuestos en puntos anteriores.

6.- Conclusiones

Tras haber realizado el trabajo, cuyos objetivos eran ver de una forma más práctica y aplicados a un problema real (en nuestro caso el perceptrón multicapa para problemas de clasificación) la paralelización con **OpenMP** y **CUDA**, hemos llegado a las siguientes conclusiones:

- El uso de la paralelización en un programa respecto al uso de un programa en serie. Como era de esperarse, paralelizar, ya sea con **OpenMP** o **CUDA**, representa una mejora bastante considerable respecto a ejecutar el programa en serie. Aplicar paralelización solo será rentable mientras mayor sea la cantidad de neuronas por capa y el número de capas, aun así para nuestro problema, incluso con topologías de pequeño tamaño, se nota el uso de paralelización (como vimos usando OpenMP).

- **OpenMP,** sencillez y eficiencia. Respecto a **OpenMP** hemos llegado a la conclusión de que si ponemos en una balanza el tiempo invertido por nosotros a la hora de implementar la paralelización y los resultados obtenidos, es muy rentable. El primer problema a tener en cuenta es la carga necesaria que vamos a tener que añadir a la paralelización, pero aun así, aunque no apliquemos la carga extra para poder paralelizar el algoritmo y únicamente paralelicemos los entrenamientos (esto se consigue usando las directivas explicadas anteriormentes en el bucle for de entrenamiento) obtendremos los resultados con una mejora de tiempo notable y sin notar una bajada de la calidad de la red entrenada. El segundo problema es más complicado de analizar, y es el uso de memoria compartida. En nuestro caso se resuelve de una forma bastante fácil, copiar los datos en estructuras independientes para más tarde volver a unir esta información.

- **CUDA** respecto a **OpenMP**. Como hemos visto en apartados anteriores, para nuestro caso en concreto, el uso de **OpenMP** para paralelizar es mejor que **CUDA**. Esto se debe a que nuestro problema ha sido afrontado basándonos en estructuras y como vimos, en **CUDA,** no se recomienda su uso, tanto por dificultad a la hora de implementarlo, como por su ineficiencia. Esto no quiere decir que **CUDA** sea peor que **OpenMP**, es decir, que para nuestro caso en concreto es peor. Sería interesante poder afrontar el problema con estructuras de memoria más simples, como son matrices, para poder hacer una comparación real del problema, y así poder comprobar si **CUDA** nos ofrece mejores resultados al disponer de más potencia de cómputo, o por el contrario se complica demasiado el problema hasta el punto de ser menos eficiente.

Como resumen global de todo el trabajo, ha sido un trabajo muy interesante porque hemos experimentado en primera persona distintos tipos de paralelización (*CPU* y *GPU*), pudiendo así entender mejor y comprender mejor el funcionamiento de la paralelización, aplicadas tanto con **OpenMP** como **CUDA**.

Aunque no hemos sido capaces de paralelizar nuestro problema en concreto con **CUDA**, hemos aprendido un camino cerrado por el que no se debe ir, de esta forma para la siguiente vez que tengamos que trabajar con **CUDA**, tendremos esto en cuenta y no usaremos estructuras en el programa, ya que son ineficientes y difíciles de implementar.
Referido a esto último hemos podido vivir una experiencia real de lo que es desarrollar un proyecto, chocar contra un problema y estudiar las posibles soluciones.

7.- Referencias

- Página de referencia para especificaciones sobre la CPU: https://www.cpu-world.com/CPUs/Zen/AMD-Ryzen%205%202600.html
- Página de referencia para especificaciones básicas sobre NVIDIA GTX 1060: https://www.nvidia.com/es-la/geforce/products/10series/geforce-gtx-1060/
- Apuntes de la asignatura sobre CUDA: https://moodle.uco.es/m2021/pluginfile.php/372323/mod_resource/content/2/COMPUTACI%C3%93N%20HETEROG%C3%89NEA%20PROGRAMACI%C3%93N%20EN%20CUDA-2.pdf
- AA.VV., 2018. *Introduction to Parallel Computing From Algorithms to Programming on State-of-the-Art Platforms*. University of Oxford. ISBN 978-3-319-98832-0.
- Imagen de ejemplo de patrones de la base de datos NotNMIST: https://enakai00.hatenablog.com/entry/2016/08/02/102917
- Documentación de cuda: https://docs.nvidia.com/cuda/
- Imagen de cuda: https://es.wikipedia.org/wiki/CUDA
- Imagen de openmp: https://www.openmp.org/
- Apuntes de algoritmos online y offline para redes neuronales de la asignatura Introducción a los modelos computacionales: https://moodle.uco.es/m2021/pluginfile.php/56709/mod_resource/content/3/IMC_Tema1_Retropropagacion.pdf
- Página con imagen de ejemplo de red neuronal : https://es.wikipedia.org/wiki/Red_neuronal_artificial