Paralelización de redes neuronales con **OpenMP** y **NVIDIA Cuda**

Arquitecturas Paralelas – 3º 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)

Paralelización de redes neuronales

Índice:

¿Qué son las **redes neuronales**?

¿Qué son las redes neuronales?

-Están formadas por neuronas.

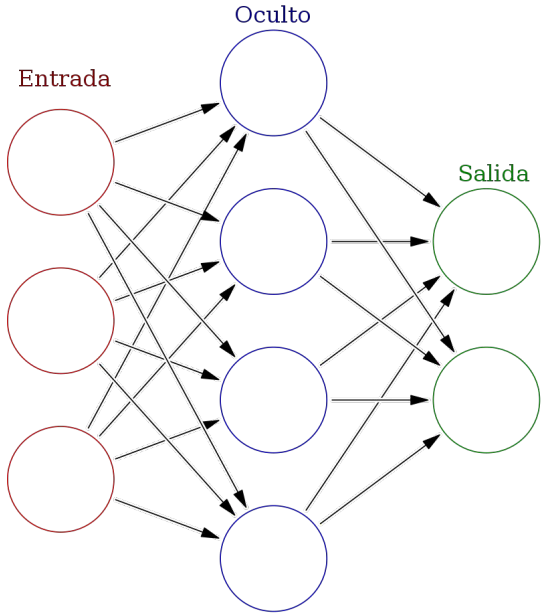
-Las neuronas se agrupan en capas.

-Las capas se dividen en: capa de entrada, capa de salida y capa oculta.

-Los enlaces entre capas tienen peso (importancia de esa conexión).

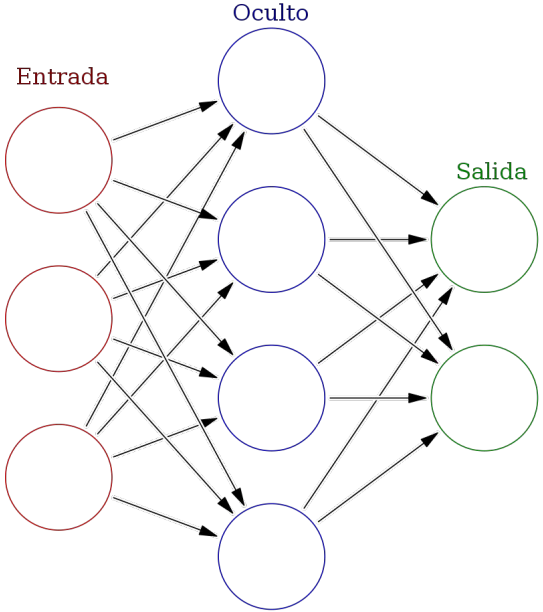
-Se dividen en inicio, entrenamiento y finalización.

-La redes neuronales se usan como sistemas expertos de apoyo a decisión.



¿Qué son las redes neuronales?

- Como es nuestra red:
  - La topología varía dependiendo del número de capas y neuronas especificado.
  - La función de activación de las neuronas es sigmoide excepto la última capa, es softMax.
  - El error usado es Entropía Cruzada, diseñada para problemas de clasificación.
  - Para el entrenamiento se aplica el algoritmo Perceptrón multicapa offline.
  - Nos basamos en un Problema de clasificación.



¿Qué son las redes neuronales?

-La base de datos que hemos utilizado es la **NotMnist**.

-900 patrones de entrenamiento.

-300 patrones de test.

- 28x28 variables de entrada (780 píxeles de la imagen)

-10 variables de salida (una por cada clase a,b,c,d,e,f,g,i y j)

¿Qué son las redes neuronales?

Algoritmo de retropropagación *off-line*

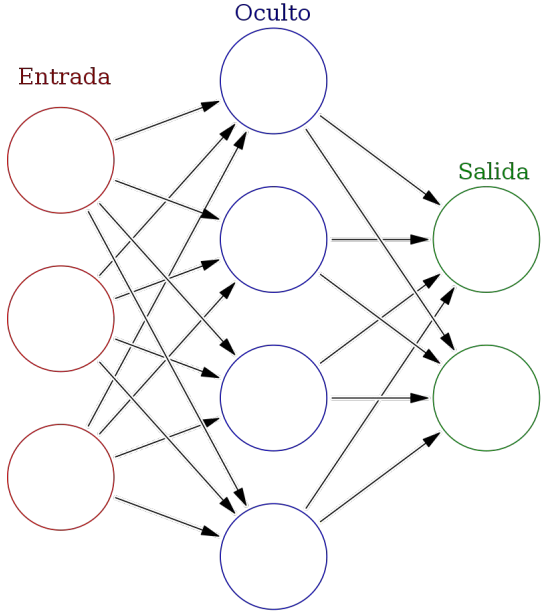
Inicio

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



Descripción de la arquitectura.

Descripción de la arquitectura.

- CPU: *AMD Ryzen 5 2600*
  - 6 cores soportando 2 hilos cada uno.

- GPU: *NVIDIA GTX 1060* (versión de 6GB)
  - Arquitectura Pascal
  - 1280 cores repartidos en 20 multiprocesadores de 64 núcleos cada uno.

Codificación del problema en serie

Entrenamiento

```
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();
	}
}
```

Las redes se basan en inicio de la topología, entrenamiento y test. Esta función sería la de entrenamiento, donde se realizan las épocas.

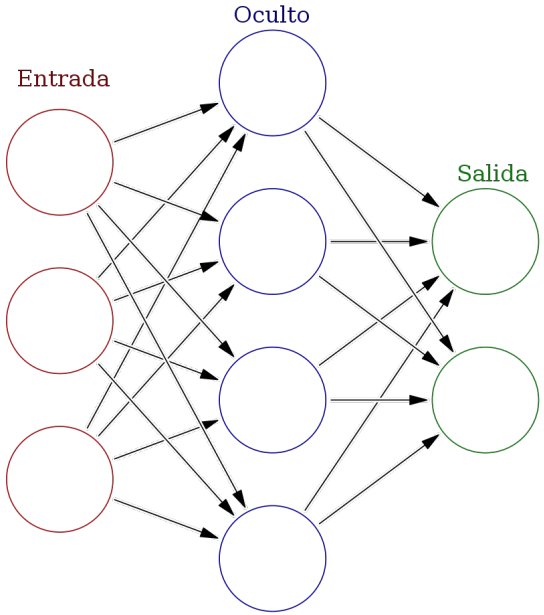
Épocas

```
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();
        }
}
```

Función de época, se basa en el algoritmo de retropropagación del error.

Resultados experimentales

- 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



Entrenamiento = Lanzar una red neuronal

Paralelización con **OpenMP**

Directivas

Nos apoyamos en dos directivas de OpenMP:

- **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 directiva paraleliza el bucle for sobre el que se escriba. se pueden añadir varias cláusulas como **nowait, private** o **public**. Nosotros usaremos principalmente **private** para definir qué variables son privadas para cada hilo.

Paralelizaciones menores

Paralelizaciones pequeñas de bucles. Por ejemplo a rellenar las estructuras, reservar o liberar memoria, etc. El tiempo de ejecución implementando estas paralelizaciones pasa de 261.724s a 259.149s.

```
#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;
```

Paralelización de los entrenamientos

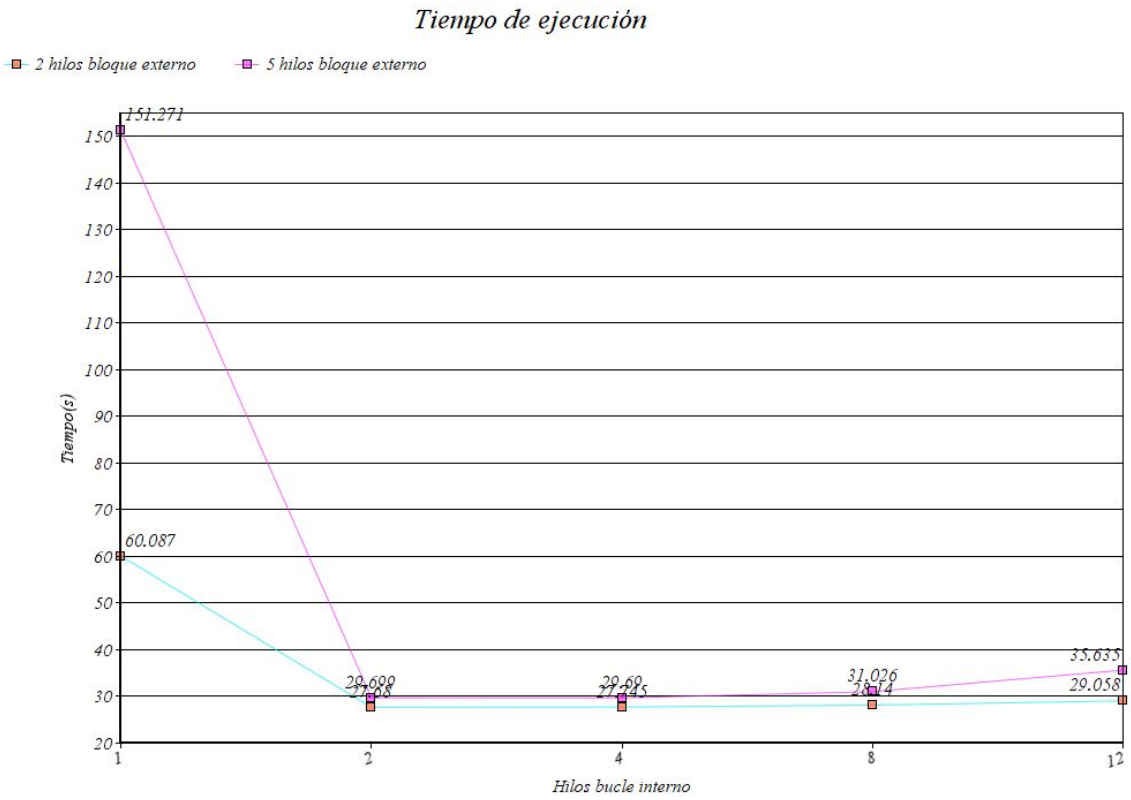
En el programa principal se realizaban 5 entrenamientos para después hacer la media y confirmar los resultados. Al paralelizar tenemos que definir el número de entrenamientos y cada uno se ejecuta en un hilo independiente. Con estas paralelizaciones obtenemos un tiempo de 133.202s.

```
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;
            }
```

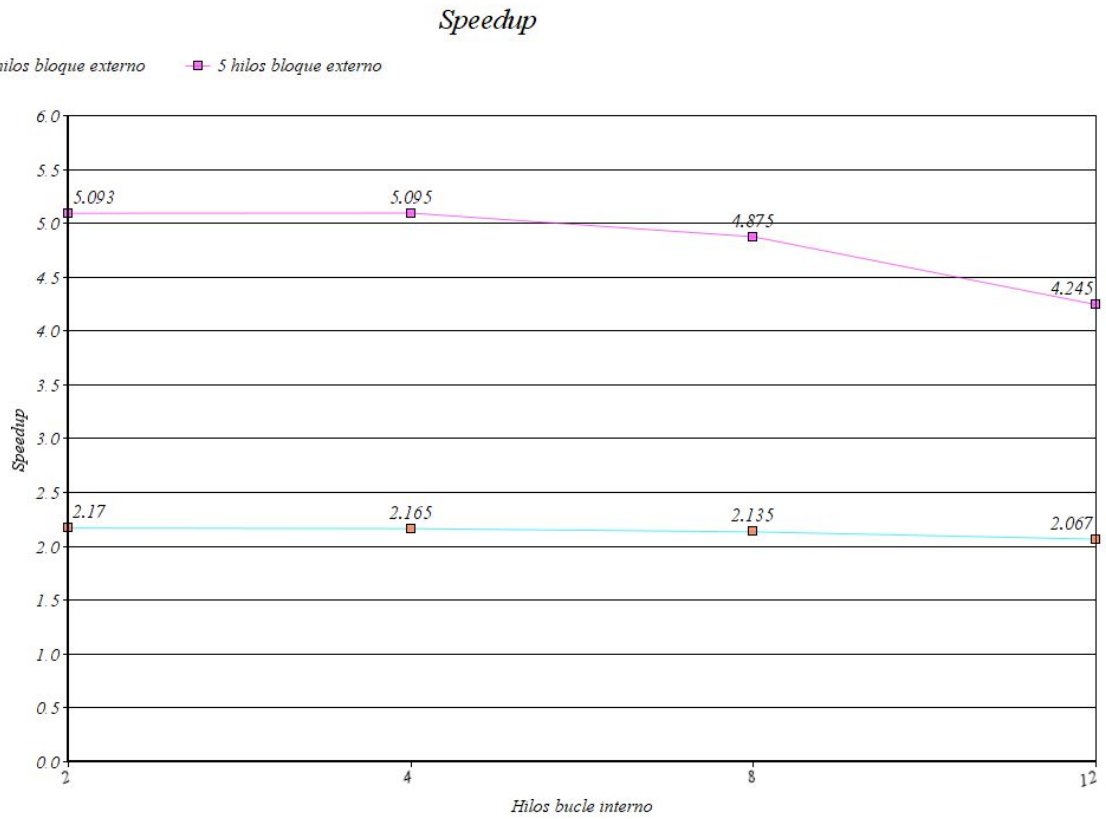
Paralelización de las épocas

Como para las épocas se calcula el ajuste patrón a patrón, podemos paralelizar este proceso dividiendo los patrones entre los hilos que le indiquemos y sumando el cambio posteriormente, para finalmente ajustar los pesos de la red. Paralelizando el algoritmo interno conseguimos un tiempo de 201.449s.

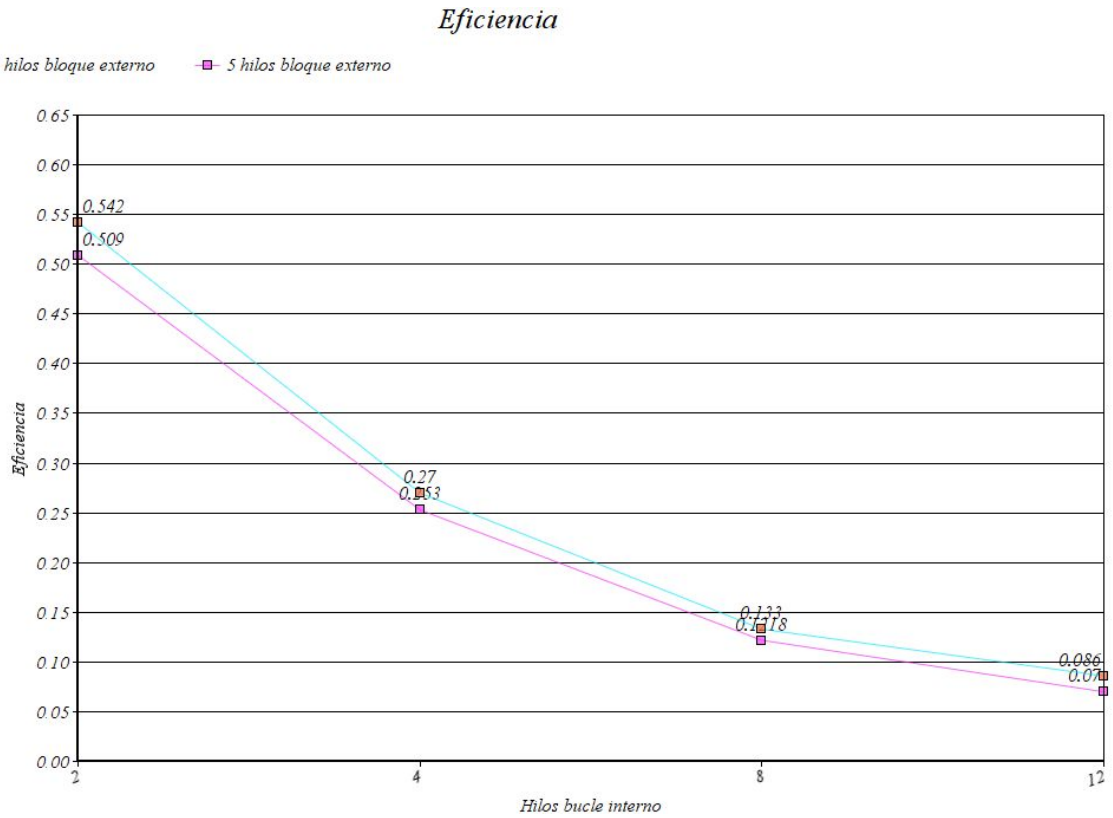
```
int j, i;
int patronesPorHilo = trainDataset->nOfPatterns/nThreads;
copyOriginalToCopys();
omp_set_num_threads(nThreads);
#pragma omp parallel for private(i,j)
for(i=0; i<nThreads; i++){
        for (j = i*patronesPorHilo; j < (i*patronesPorHilo)+patronesPorHilo; j++) {
                performEpoch(trainDataset->inputs[j],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

if (!online) {
        weightAdjustment();
}
```



Gráfica de tiempos.



Gráfica de speedup.



Gráfica de eficiencia.

Paralelización con **NVIDIA Cuda**

Conocer la capacidad de cómputo de nuestra GPU

```
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  }
```

Conocer la capacidad de cómputo de nuestra GPU

Dependiendo de la propiedad major y minor entonces la arquitectura será:

- 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.

Kernel

```
#include <cuda_runtime.h>

#include <stdio.h>

__global__ void cudahello(){
    int thread = threadIdx.x;
    int block = blockIdx.x;
    printf("Hola Mundo! Soy el hilo %d del bloque %d\n", thread, block);
}

int main(){
    cudahello<<<4,4>>>();
    cudaDeviceSynchronize();
}
```

Device (Kernel)

Número de bloques

Hilos por bloque

Host

El máximo de hilos por bloque son 1024, aunque se recomienda usar múltiplos de 32 para usar la memoria compartida que ofrecen los warps. Respecto a nuestra arquitectura se usarían 32 hilos o 64 hilos.

Problemas para implementar nuestro código

Al intentar implementar nuestro programa con CUDA nos encontramos un problema el cual no tenemos tiempo a solucionar, pero del que sí hemos aprendido y sabemos como solucionar. El paso de memoria de un sistema a otro no es tan básico como podría parecer al utilizar estructuras.

Nuestro programa se basa en estructuras con memoria dinámica en su interior, lo cual hace muy complejo el paso de estas del host al dispositivo.

Solución

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

```
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'
```

Solución

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

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

```
// 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);
```

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

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

Conclusiones

Conclusiones

- **OpenMP:** es muy fácil de implementar si conoces el problema de la memoria compartida. En nuestro caso, copiar la información para no tener problemas.
- **CUDA:** no podemos hablar de la implementación del problema en CUDA pero sí de cómo funciona la memoria para futuros proyectos. Hay que entender bien cómo se copia la memoria del host al device para poder trabajar eficientemente y aprovechar completamente la capacidad de cómputo.
- Renta en general aplicar paralelización, sobre todo si requiere de cierta exigencia de cómputo.

Bibliografía:

- 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