

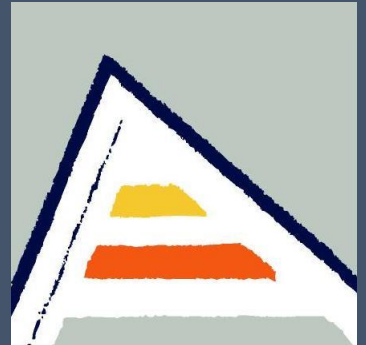
ARQUITECTURA DE LOS COMPUTADORES

PRÁCTICA 4

*FASE 4: Implementación de una rutina con
CUDA para la aceleración de algoritmos
utilizando GPGPU*

INTEGRANTES DEL GRUPO

Pavel Razgovorov | Y0888160Y (Controlador)
María Rico Martínez | 48775095F
Eddie Rodríguez Pastor 74391601X (Director)
Miguel Sánchez Moltó | 20096774H
César Sarrión Posas | 20456867T (Secretario)



27.05.2016

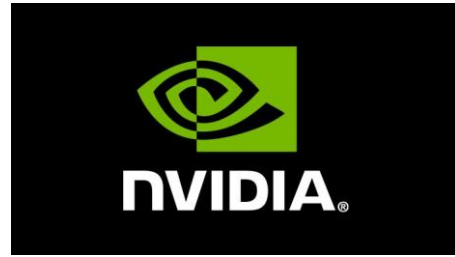
Grupo: 04

ÍNDICE

- CUDA.....Página 2
- IV. Clasificador paralelo basado en SOM.....Página 7
- Algoritmo.....Página 10
- Ejecución del algoritmo.....Página 12
- Problemas encontrados.....Página 13
- Referencias.....Página 14

¿Qué es CUDA?

CUDA es una arquitectura de cálculo paralelo de NVIDIA que aprovecha la gran potencia de la GPU (unidad de procesamiento gráfico) para proporcionar un incremento extraordinario del rendimiento del sistema.



Gracias a millones de GPUs CUDA vendidas hasta la fecha, miles de desarrolladores, científicos e investigadores están encontrando innumerables aplicaciones prácticas para esta tecnología en campos como el procesamiento de vídeo e imágenes, la biología y la química computacional, la simulación de la dinámica de fluidos, la reconstrucción de imágenes de TC, el análisis sísmico o el trazado de rayos, entre otras.

Ventajas

CUDA presenta ciertas ventajas sobre otros tipos de computación sobre GPU utilizando APIs gráficas.

- Lecturas dispersas: se puede consultar cualquier posición de memoria.
- Memoria compartida: CUDA pone a disposición del programador un área de memoria de 16KB (o 48KB en la serie Fermi) que se compartirá entre threads. Dado su tamaño y rapidez puede ser utilizada como caché.
- Lecturas más rápidas de y hacia la GPU.
- Soporte para enteros y operadores a nivel de bit.

Limitaciones

- No se puede utilizar recursividad, punteros a funciones, variables estáticas dentro de funciones o funciones con número de parámetros variable
- No está soportado el renderizado de texturas
- En precisión simple no soporta números desnormalizados o NaNs
- Puede existir un Cuello de botella entre la CPU y la GPU por los anchos de banda de los buses y sus latencias.
- Los threads o Hilo de ejecución, por razones de eficiencia, deben lanzarse en grupos de al menos 32, con miles de hilos en total.

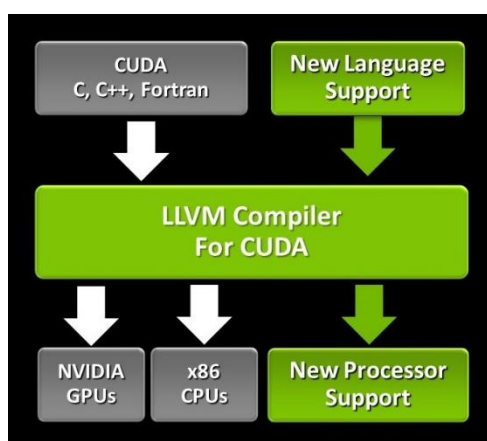
Jerarquía de memoria

- Los hilos en CUDA pueden acceder a distintas memorias, unas compartidas y otras no.
- En primer lugar, está la memoria privada de cada hilo, solamente accesible desde él mismo.
- Cada bloque de hilos posee también un espacio de memoria, compartida en este caso por los hilos del bloque y con un ámbito de vida igual que el del propio bloque.
- Todos los hilos pueden acceder a una memoria global.

Procesamiento paralelo con CUDA

Los sistemas informáticos están pasando de realizar el “procesamiento central” en la CPU a realizar “coprocesamiento” repartido entre la CPU y la GPU. Para posibilitar este nuevo paradigma computacional, NVIDIA ha inventado la arquitectura de cálculo paralelo CUDA, que ahora se incluye en las GPUs GeForce, ION Quadro y Tesla GPUs, lo cual representa una base instalada considerable para los desarrolladores de aplicaciones.

En el mercado de consumo, prácticamente todas las aplicaciones de vídeo se han acelerado, o pronto se acelerarán, a través de CUDA, como demuestran diferentes productos de Elemental Technologies, MotionDSP y LoiLo, Inc.



CUDA ha sido recibida con entusiasmo por la comunidad científica. Por ejemplo, se está utilizando para acelerar AMBER, un simulador de dinámica molecular empleado por más de 60.000 investigadores del ámbito académico y farmacéutico de todo el mundo para acelerar el descubrimiento de nuevos medicamentos.

En el mercado financiero, Numerix y CompatibL introdujeron soporte de CUDA para una nueva aplicación de cálculo de

riesgo de contraparte y, como resultado, se ha multiplicado por 18 la velocidad de la aplicación. Cerca de 400 instituciones financieras utilizan Numerix en la actualidad.

Un buen indicador de la excelente acogida de CUDA es la rápida adopción de la GPU Tesla para aplicaciones de GPU Computing. En la actualidad existen más de 700 clusters de GPUs instalados en compañías Fortune 500 de todo el mundo, lo que incluye empresas como Schlumberger y Chevron en el sector energético o BNP Pariba en el sector bancario.

Por otra parte, la reciente llegada de los últimos sistemas operativos de Microsoft y Apple (Windows 8 y Snow Leopard) está convirtiendo el GPU Computing en una tecnología de uso masivo. En estos nuevos sistemas, la GPU no actúa únicamente como procesador gráfico, sino como procesador paralelo de propósito general accesible para cualquier aplicación.

La plataforma de cálculo paralelo CUDA® proporciona unas cuantas extensiones de C y C++ que permiten implementar el paralelismo en el procesamiento de tareas y datos con diferentes niveles de granularidad. El programador puede expresar ese paralelismo mediante diferentes lenguajes de alto nivel como C, C++ y Fortran o mediante estándares abiertos como las directivas de OpenACC.



En todo programa realizado en CUDA, podemos distinguir dos partes:

- **Código Host en la CPU que hace interfaz con la GPU.**

En el nivel host, existen dos APIs

- **Runtime:** Simplificada, más sencilla de usar.
- **Driver:** más flexible, más compleja de usar. (La versión driver no implica más rendimiento, sino más flexibilidad a la hora de trabajar con la GPU)

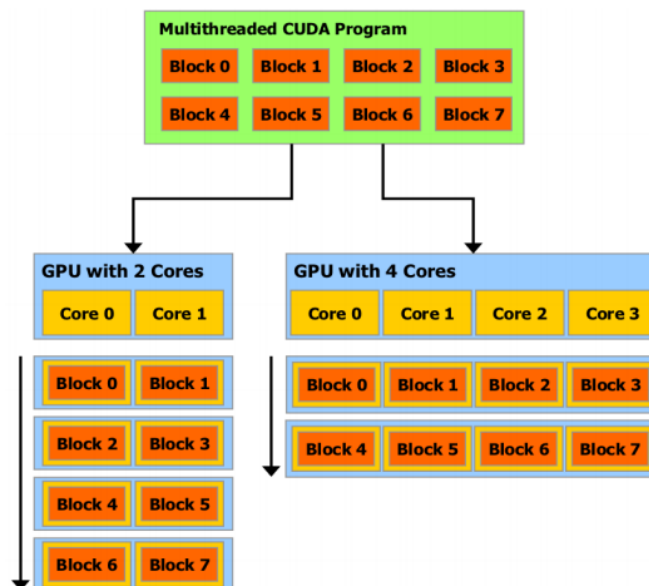
- **Código Kernel que se ejecuta sobre la GPU.**

Así pues, como conceptos básicos necesarios, destacamos:

- **Bloque**

Los mismos bloques, se dividen de forma óptima según el número de núcleos del sistema.

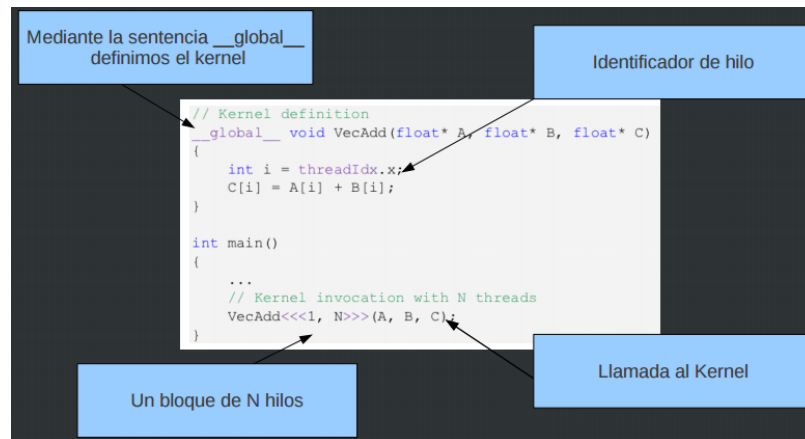
- Es una agrupación de hilos
- Cada bloque se ejecuta sobre un solo SM
- Un SM puede tener varias asignaciones de varios bloques.



- **Kernel**

Se trata de una función la cual al ejecutarse lo hará en N distintos hilos en lugar de en secuencial.

El modelo de programación de CUDA asume que los bloques que ejecutan los kernels se ejecutarán en la GPU mientras que el resto se ejecutará en la CPU.

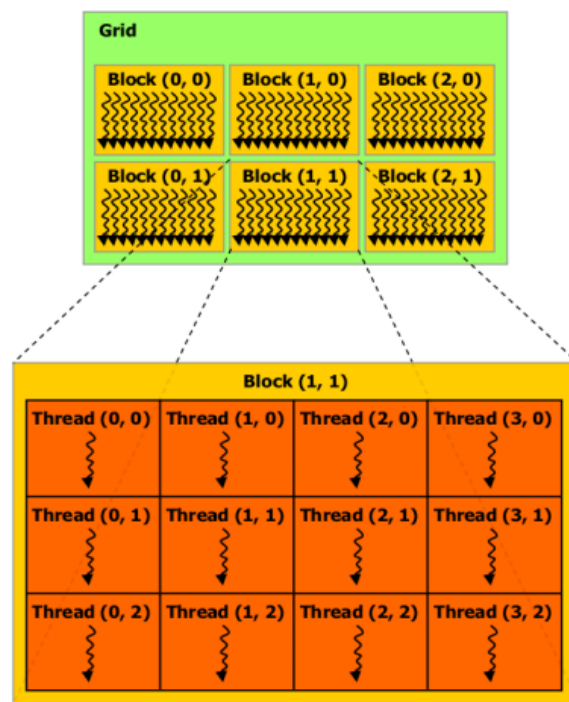


Tanto la GPU como la CPU mantienen su propio espacio de memoria DRAM, con lo que en tiempo de ejecución se deben gestionar las transferencias entre los dos espacios de memoria.

- **Grid**

Será la forma de estructurar los bloques en el kernel.

- Bloques (de una o dos dimensiones)
- Hilos/Bloque (de una, dos o tres dimensiones)



De esta manera, en la ejecución en CUDA podemos destacar que cada uno de los bloques, se divide en warps de 32 hilos, ejecutándose cada uno de estos en paralelo. Además, cabe destacar el aprovechamiento del pipeline GPU y la ejecución de forma paralela realizada sobre los 32 núcleos del SM.

IV. Clasificador paralelo basado en SOM

En esta práctica, realizaremos un problema el cual consiste en implementar en CUDA una versión simplificada de un mapa auto-organizativo, concretamente una red neuronal basada en SOM. Este proyecto, permite la clasificación de diferentes patrones de datos de entrada.

Esta red, constará de dos modos:

- Modo de entrenamiento: permitirá configurar la red.
- Modo de clasificación: la cual utilizará el entrenamiento previo para clasificar los datos que reciba como entrada, centrándose el problema exclusivamente en esta fase.

La configuración del SOM consistirá en una colección de neuronas o nodos, los cuales contienen un vector de pesos con las mismas dimensiones que los patrones recibidos como entrada. Cada nodo, contiene individualmente una etiqueta como salida de la clasificación, apareciendo esta red neuronal, dispuesta en forma de rejilla rectangular o grid.

Así pues, en los siguientes apartados, procederemos a explicar detalladamente esta tipología de red de mapas auto-organizados:

Introducción

En 1982 T. Kohonen presentó un modelo de red denominado mapas auto-organizados o SOM (Self-Organizing Maps), basado en ciertas evidencias descubiertas a nivel cerebral.



Este tipo de red posee un aprendizaje no supervisado competitivo.

No existe ningún maestro externo que indique si la red neuronal está operando correcta o incorrectamente porque no se dispone de ninguna salida objetivo hacia la cual la red neuronal deba tender.

La red auto-organizada debe descubrir rasgos comunes, regularidades, correlaciones o categorías en los datos de entrada, e incorporarlos a su estructura interna de conexiones.

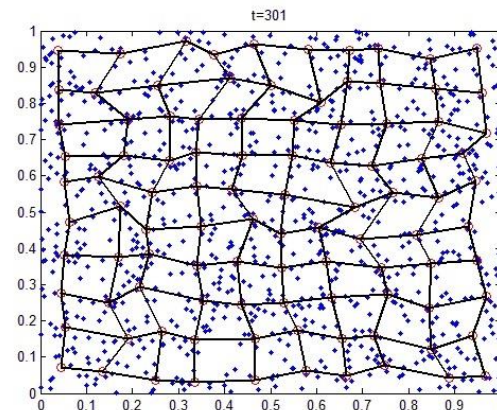
Se dice, por tanto, que las neuronas deben auto-organizarse en función de los estímulos (datos) procedentes del exterior. En el aprendizaje competitivo las neuronas compiten unas con otras con el fin de llevar a cabo una tarea dada.

Se pretende que cuando se presente a la red un patrón de entrada, sólo una de las neuronas de salida (o un grupo de vecinas) se active.

Por tanto, las neuronas compiten por activarse, quedando finalmente una como neurona vencedora y el resto anuladas, que son forzadas a sus valores de respuesta mínimos.

El objetivo de este aprendizaje es categorizar los datos que se introducen en la red. Se clasifican valores similares en la misma categoría y, por tanto, deben activar la misma neurona de salida.

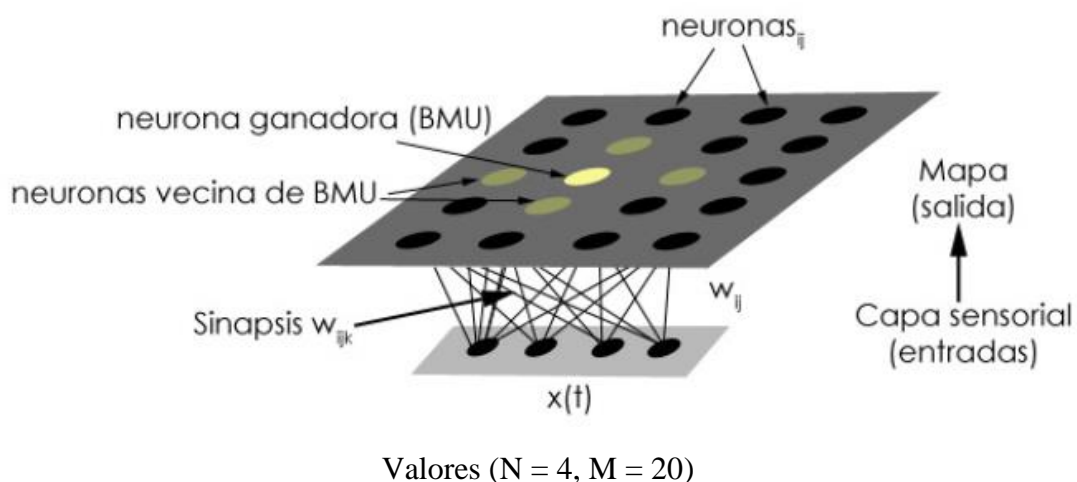
Las clases o categorías deben ser creadas por la propia red, puesto que se trata de un 1 aprendizaje no supervisado, a través de las correlaciones entre los datos de entrada.



Arquitectura SOM

Un modelo SOM está compuesto 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. Normalmente, las neuronas de la capa de salida se organizan en forma de mapa bidimensional como se muestra en la figura:



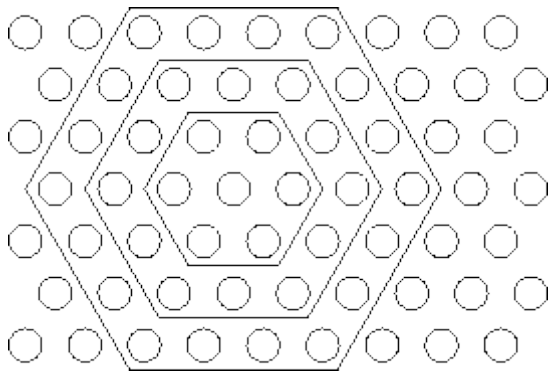
Así pues, las conexiones entre las dos capas que forman la red son siempre hacia delante, es decir, la información se propaga desde la capa de entrada hacia la capa de salida.

Cada neurona de entrada i está conectada con cada una de las neuronas de salida j mediante un peso w_{ji} .

De esta forma, las neuronas de salida tienen asociado un vector de pesos W_j llamado vector de referencia, debido a que constituye el vector prototipo de la categoría representada por la neurona de salida j .

Así, 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, puede decirse que existen conexiones laterales de excitación e inhibición implícitas, pues aunque no estén conectadas, cada una de estas neuronas va a tener cierta influencia sobre sus vecinas.



Así pues, tal y como se observa en la imagen, de esa manera se podría representar “la vecindad” entre neuronas dada una ganadora.

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. Las neuronas adyacentes pertenecen a una vecindad N_j de la neurona j . La topología y el número de neuronas permanecen fijos desde el principio.

El número de neuronas determina la suavidad de la proyección, lo cual influye en el ajuste y capacidad de generalización del SOM.

Durante la fase de entrenamiento, el SOM forma una red elástica que se pliega dentro de la nube de datos originales. El algoritmo controla la red de modo que tiende a aproximar la densidad de los datos.

Los vectores de referencia del codebook se acercan a las áreas donde la densidad de datos es alta. Eventualmente unos pocos vectores el codebook están en áreas donde existe baja densidad de datos.

El algoritmo del SOM

El proceso de aprendizaje del SOM es el siguiente:

- **PASO 1**

Un vector x es seleccionado al azar del conjunto de datos y se calcula su distancia (similitud) a los vectores del codebook, usando, por ejemplo, la distancia euclídea:

$$\|x - m_c\| = \min_j \{\|x - m_j\|\}$$

- **PASO 2**

Una vez que se ha encontrado el vector más próximo o BMU (best matching unit) el resto de vectores del codebook es actualizado. El BMU y sus vecinos (en sentido topológico) se mueven cerca del vector x en el espacio de datos.

La magnitud de dicha atracción está regida por la tasa de aprendizaje. Mientras se va produciendo el proceso de actualización y nuevos vectores se asignan al mapa, la tasa de aprendizaje decrece gradualmente hacia cero.

Junto con ella también decrece el radio de vecindad también. La regla de actualización para el vector de referencia dado i es la siguiente:

$$m_j(t+1) = \begin{cases} m_j(t) + \alpha(t)(x(t) - m_j(t)) & j \in N_c(t) \\ m_j(t) & j \notin N_c(t) \end{cases}$$

Los pasos 1 y 2 se van repitiendo hasta que el entrenamiento termina. El número de pasos de entrenamiento se debe fijar antes a priori, para calcular la tasa de convergencia de la función de vecindad y de la tasa de aprendizaje.

Una vez terminado el entrenamiento, el mapa ha de ordenarse en sentido topológico: n vectores topológicamente próximos se aplican en n neuronas adyacentes o incluso en la misma neurona.

Algoritmo a implementar: ClasificacionSOMGPU

```
int ClasificacionSOMGPU() {  
  
    //TSOM SOM;  
    //TPatrones Patrones;  
    int * d_EtiquetaGPU = (int*)malloc(Patrones.Cantidad*sizeof(int));  
  
    //COPIA DE LA STRUCT D_AUX  
    d_AUX aux;  
    aux.Alto = SOM.Alto;  
    aux.Ancho = SOM.Ancho;  
    aux.Dimension = SOM.Dimension;  
    aux.d = maxDepth();  
  
    int size = 0;  
    for (int x = 0; x < SOM.Alto; x++){  
        for (int y = 0; y < SOM.Ancho; y++){  
            size += sizeof(SOM.Neurona[x][y].pesos);  
        }  
    }  
  
    // COPIA DE LOS PESOS  
  
    //float* pesos = (float*)malloc(size);  
    float* pesos = (float*)malloc((maxDepth()*SOM.Alto*SOM.Ancho)*4);  
  
    for (int i = 0; i < SOM.Alto; i++){  
        for (int j = 0; j < SOM.Ancho; j++){  
            for (int k = 0; k < sizeof(SOM.Neurona[i][j].pesos)/4; k++){  
                pesos[getIndex(i, j, k)] = SOM.Neurona[i][j].pesos[k];  
            }  
        }  
    }  
  
    cudaMalloc(&aux.pesos, (maxDepth()*SOM.Alto*SOM.Ancho) * 4);  
    cudaMemcpy(aux.pesos, pesos, (maxDepth()*SOM.Alto*SOM.Ancho) * 4,  
        cudaMemcpyHostToDevice);  
  
    //COPIA DE LABELS  
    int* labels = (int*)malloc(sizeof(int) * SOM.Alto * SOM.Ancho);  
  
    for (int i = 0; i < SOM.Ancho; i++){  
        for (int j = 0; j < SOM.Alto; j++){  
            labels[getIndex(i, j)] = SOM.Neurona[i][j].label;  
        }  
    }  
  
    cudaMalloc(&aux.label, (sizeof(int) * SOM.Alto * SOM.Ancho));  
    cudaMemcpy(aux.label, labels, (sizeof(int) * SOM.Alto * SOM.Ancho),  
        cudaMemcpyHostToDevice);  
  
    //COPIA DE D_PATRONES  
  
    d_TPatrones d_Patrones;  
  
    d_Patrones.Cantidad = Patrones.Cantidad;  
    d_Patrones.Dimension = Patrones.Dimension;  
  
    float* pesosP =(float*)malloc(Patrones.Cantidad*Patrones.Dimension*4);
```

```

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

cudaMalloc(&d_Patrones.Pesos, Patrones.Cantidad*Patrones.Dimension*4);
cudaMemcpy(d_Patrones.Pesos, pesosP, Patrones.Cantidad *
Patrones.Dimension*4, cudaMemcpyHostToDevice);

float* d_pesos;
cudaMalloc(&d_pesos, (maxDepth()*SOM.Alto*SOM.Ancho) * 4);
cudaMemcpy(d_pesos, pesos, sizeof(pesos), cudaMemcpyHostToDevice);

//COPIA ETIQUETAGPU
cudaMalloc(&d_EtiquetaGPU, Patrones.Cantidad*sizeof(int));

dim3 block(16);
dim3 grid((Patrones.Cantidad + (block.x - 1)) / block.x);

calculadorNeuronal<<<grid,block>>>(d_EtiquetaGPU, aux, d_Patrones);

cudaMemcpy(EtiquetaGPU, d_EtiquetaGPU, Patrones.Cantidad*sizeof(int),
cudaMemcpyDeviceToHost);

cudaError_t cudaerr = cudaDeviceSynchronize();
if (cudaerr != CUDA_SUCCESS) printf("kernel launch failed with error
\"%s\".\n", cudaGetErrorString(cudaerr));
printf("End\n");

for (int i = 0; i < 1024; i++){
    printf("Final %i: %i\n", i, EtiquetaGPU[i]);
    //EtiquetaGPU[i] = EtiquetaCPU[i];
}


return OKCLAS;
}

```

Análisis de los resultados

Así pues, a continuación, procederemos a realizar una comparativa sobre el tiempo de ejecución que han experimentado dos equipos escogidos, distintos entre sí, para el algoritmo diseñado.

De esta manera, las características de los equipos empleados son las siguientes:



PC 1

- Sistema operativo: Windows 10 ~ 32 bits
- Procesador: i5 4460 Quad Core @ 3'3 Ghz
- Número de núcleos: 4
- Memoria RAM: 14GB @ 1600MHz
- Tarjeta gráfica: NVIDIA GTX 750 OC 2GB



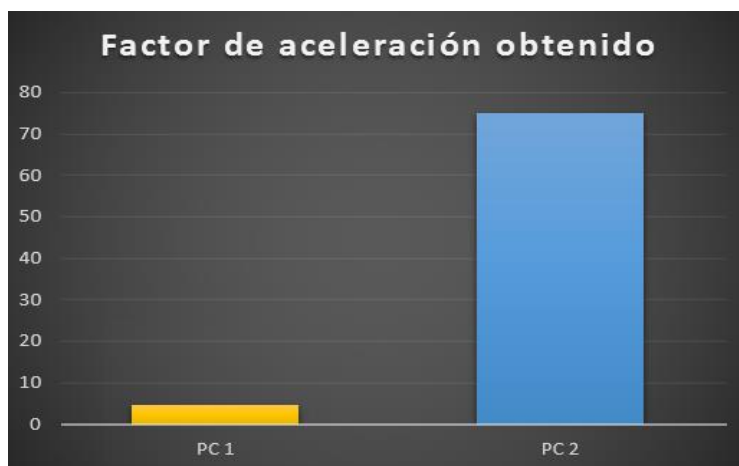
PC 2

- Sistema operativo: Windows 8.1 ~ 64 bits
- Procesador: Intel® Core™ i7-4700HQ @ 2.40 GHz
- Número de núcleos: 8
- Memoria RAM: 8GB
- Tarjeta gráfica: NVIDIA GeForce GTX 750

Así pues, los resultados obtenidos sobre el factor de aceleración para los dos equipos testeados, resultan ser los siguientes:

	Aceleración
PC 1	4,54772x
PC 2	75,087648x

De esta manera, la gráfica para dichos resultados resulta de la siguiente manera:



Así pues, de esta manera podemos observar una clara diferencia entre los factores de aceleración obtenidos para cada uno de los equipos testeados, siendo el claro vencedor el “PC 2”. Éste, al presentar un mayor factor de aceleración prueba que la paralelización experimentada en dicho equipo ha resultado mucho más provechosa y fructífera que en el “PC 1”.

Problemas encontrados

El principal problema encontrado, ha sido la imposibilidad de copiar arrays de más de una dimensión, optando de esa manera por trabajar sólo con una dimensión.

De esta manera, al ejecutar el algoritmo en nuestro equipo, se comprueba que no funciona correctamente, pues se produce una violación de segmento que impide que termine correctamente la llamada al kernel.

Referencias

■ **CUDA**

<http://dis.um.es/~domingo/apuntes/AlgProPar/1112/CUDA.pdf>

■ **Neuronas y conexiones**

<http://perso.wanadoo.es/alimanya/funcion.htm>

■ **Mapas auto-organizados de Kohonen (SOM)**

<http://halweb.uc3m.es/esp/Personal/personas/jmmarin/esp/DM/tema5dm.pdf>

