Universidad de Valladolid

Escuela Técnica Superior de Ingenieros de Telecomunicación

Grado en Tecnologías de Telecomunicación

Implementación de una red neuronal tipo perceptrón en GPU

Presentado por:

Roberto Torre Arranz

Tutelado por:

Mario Martínez Zarzuela

Valladolid, 02 de Septiembre de 2015

Resumen

El objetivo de este trabajo de fin de grado es implementar un perceptrón multicapa en la plataforma de computación en GPU de propósito general CUDA, para poder explotar el paralelismo por el que se caracterizan las tarjetas gráficas.

Se ha realizado tanto una tarea de formación en el ámbito de las redes neuronales básicas, hasta llegar al perceptrón multicapa, como en el ámbito del lenguaje de programación CUDA, del que se partía desde cero. Así mismo, se ha aprendido a utilizar de forma exitosa la biblioteca cuBLAS, especializada en operaciones matemáticas.

Adicionalmente, se ha podido comprobar cómo, con las pruebas realizadas sobre la multiplicación de matrices, operación base del perceptrón multicapa, la diferencia de eficiencia entre una GPU y una CPU utilizando CUDA puede ser hasta 20 veces mayor medida en GFLOPS, mientras que, si utilizamos la biblioteca cuBLAS, obtenemos un rendimiento hasta 1.000 veces mayor.

Palabras clave

Redes neuronales, perceptrón multicapa, multiplicación de matrices, lenguaje CUDA, programación en GPU.

Abstract

The aim of this master thesis is to implement an efficient multilayer perceptron in the GPU general purpose platform CUDA, in order to exploit the parallelism offered by graphics cards.

Both training neural networks, until multilayer perceptron, and CUDA language has been done. Moreover, cuBLAS library, a math library from CUDA language has been correctly learned.

Efficiency between GPU and CPU using CUDA language has been successfully checked by performing simulations on matrix multiplication, which is the base of multilayer perceptron. The performance on GPU can be 20 times higher than CPU in terms of GFLOPS, meanwhile, if you use cuBLAS library, the performance can be 1.000 times higher.

Keywords

Neurnal networks, multilayer perceptron, matrix multiplication, CUDA language, GPU programming.

Agradecimientos

A mi familia y amigos, por haberme apoyado todo este tiempo, a mi tutor, por darme la posibilidad de introducirme en el mundo de la investigación en rendes neuronales, y a mis compañeros de laboratorio, por haber hecho en una estancia de trabajo un ambiente cálido y acogedor.

Contenido

1. Introducción, Motivación y Objetivos

Las redes neuronales son en estos momentos uno de los campos de investigación en auge. El deseo del ser humano por intentar lograr algo parecido a un cerebro crece con el tiempo, y cuanto más se consigue avanzar en la investigación, más crece ese deseo.

En este trabajo se realizará una red neuronal artificial básica, como es el perceptrón multicapa, desarrollado sobre un lenguaje de programación en GPU de propósito general CUDA, desarrollado sobre una base de C++. A su vez, se hará un estudio del rendimiento y una comparativa entre CPU y GPU para nuestro perceptrón, en particular, se hará un estudio del rendimiento para la multiplicación de matrices, operación básica del perceptrón a la hora del cálculo.

El objetivo de este trabajo es el de demostrar que se puede realizar un perceptrón en este lenguaje de programación, y que puede ser mucho más rápido que en otros lenguajes como C++, ya que CUDA es capaz de explotar el paralelismo de la tarjetas gráficas, algo que C++ no es capaz de hacer. Además, se utilizará una librería auxiliar denominada cuBLAS, de cálculo matemático, que hará que nuestro rendimiento sea no mucho, sino muchísimo mayor que utilizando un programa simple en C++.

El trabajo constará de una parte teórica de explicación de los conceptos a tratar, empezando por las redes neuronales, desde el inicio hasta el perceptrón multicapa; y continuando por los programas realizados en CUDA, ya que ha sido necesaria la comprensión de este lenguaje de programación antes de hacer ejercicios más avanzados. Posteriormente, se concluirá con la explicación del código de todos los ejercicios realizados tanto en CUDA como en cuBLAS y del perceptrón multicapa híbrido realizado en C++ y CUDA. Además, también se hablará de las pruebas realizadas en las simulaciones de multiplicación de matrices y del rendimiento obtenido en cada uno de los posibles casos estudiados. Finalmente, se adjunta un presupuesto del coste del trabajo realizado, las líneas futuras a seguir y las conclusiones del proyecto.

En el apéndice se muestran los códigos realizados y de los que se habla durante el trabajo y una colección de ejercicios propuestos para alumnos que se inicien en este tema, útil en las primeras fases del aprendizaje.

2. Las Redes Neuronales

Una de las grandes preocupaciones del ser humano a lo largo de la naturaleza ha sido la comprensión de nuestro cerebro, la diferencia con el resto de los animales. Este enigma está asociado con el de la inteligencia. Es por eso que a lo largo de la historia se han creado múltiples teorías sobre la composición de ese "algo" que tenemos los humanos que nos diferencia del resto de seres vivos. A medida que se ha ido avanzando el estudio ha ido mejorando y se han podido ir descubriendo y comprobando hipótesis sobre nuestro sistema nervioso y nuestra inteligencia. En los últimos años, uno de los mayores objetivos del hombre es el de emular mediante máquinas la inteligencia del ser humano. Éste es el objetivo de la disciplina científica conocida como Inteligencia Artificial.

Las Redes Neuronales Artificiales emulan las redes de neuronas humanas con el objetivo de diseñar elementos neuronales de procesamiento paralelo. La investigación de estas redes se encuentra dentro de la disciplina de la inteligencia artificial, y dentro de ésta, en la subdisciplina conocida como la perspectiva subsimbólica. Esta perspectiva trata de estudiar los mecanismos

físicos que nos capacitan como seres inteligentes, frente a los programas de ordenador que son demasiado simples y predecibles. Trata de estudiar los mecanismos de los sistemas nerviosos, del cerebro, además de su estructura y características lógicas. (Martín del Brío, Molina 2006)

2.1. Fundamentos biológicos

A nivel biológico, el sistema nervioso de un ser humano se compone de alrededor de 100.000 millones de neuronas, conectadas cana una de ellas con unas 10.000. Las neuronas se componen de tres partes, el axón, que es una ramificación de salida de la neurona, las dendritas, que son ramificaciones de entrada que propagan la señal al interior de la neurona y suelen conectarse con los axones del resto de neuronas y el núcleo, donde se procesa la información. Como se puede observar, estas tres partes se parecen bastante a cualquier sistema electrónico, con una entrada, una salida y un núcleo de procesamiento. (Florez López, Fernández Fernández 2008)

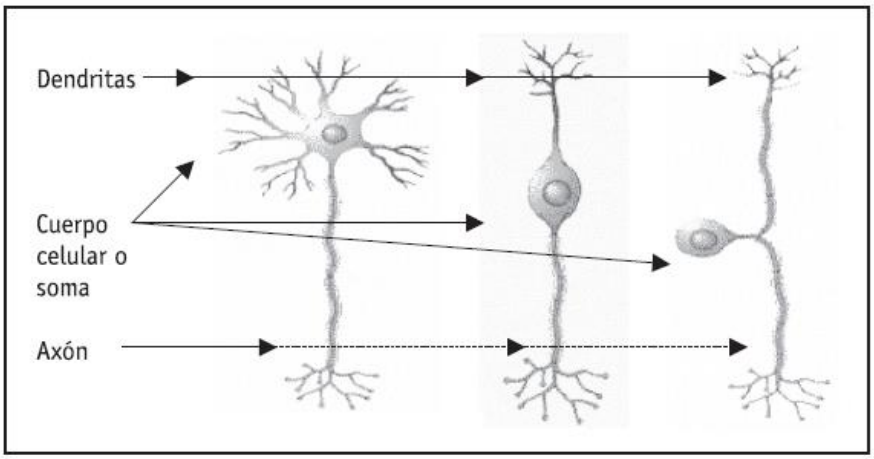


Figura 2.1. Estructura de una neurona (Florez López, Fernández Fernández 2008)

Sin embargo, construir un sistema así es prácticamente imposible a día de hoy. En la tabla 2.1 podemos observar las diferencias entre un sistema neuronal humano y el de un ordenador. Se puede observar como aún estamos muy lejos de poder emular un cerebro humano, pero poco a poco se van dando pequeños pasos hacia delante. (Florez López, Fernández Fernández 2008)

Características	Cerebro humano	Computador
Velocidad de proceso	Entre 10^{-3} y 10^{-2} seg.	Entre 10^{-8} y 10^{-9} seg.
Estilo de procesamiento	Paralelo	Secuencial (en serie)
Número de procesadores	Entre 10^{11} y 10^{14}	Pocos
Conexiones	10.000 por procesador	Pocas
Almacenamiento del conocimiento	Distribuido	En direcciones fijas (posiciones precisas)
Tolerancia a fallos	Amplia	Poca o nula
Tipo de control del proceso	Autoorganizado (democrático)	Centralizado (dictatorial)
Consumo de energía para ejecutar una operación/sg.	10^{-16} Julios	10^{-6} Julios

Tabla 2.1. Comparativa Cerebro Humano – Computador (Florez López, Fernández Fernández 2008)

2.2. Historia de las redes neuronales artificiales

Las primeras investigaciones sobre Redes Neuronales tienen lugar a principios del siglo XX. La primera implementación de una red fue un dispositivo hidráulico realizado por Russel. Sin embargo, no fue hasta la década de los 40 cuando este campo cobró una gran fuerza. En esta década, el neurofísico Warren McCullch y el matemático Walter Pitts realizaron el primer modelo matemático de una red neuronal artificial en el libro *Embodiments of Mind (1965).* Posteriormente, Hebb desarrolló una regla denominada "Regla de Hebb" que se convirtió en la primera regla de aprendizaje de las neuronas, y una predecesora de los algoritmos que hay hoy en día.

A partir de estas aportaciones iniciales, en los años 50 se produjo una verdadera explosión en este campo de investigación. Marvin Minsky obtuvo los primeros resultados prácticos en redes neuronales, y en 1951 entre Minsky y Edmons lograron diseñar una máquina con 40 neuronas que se ajustaba a la realización de ciertas tareas de forma correcta. La máquina estaba construida con tubos, motores y relés, y fue capaz de modelar el comportamiento de una rata buscando comida en un laberinto.

Más tarde, Albert Uttley comenzó a desarrollar nuevos paradigmas de redes, creando una máquina compuesta por separadores lineales que ajustaba sus parámetros de entrada utilizando la medida de entropía de Shannon.

En 1957, Frank Rosenblatt generalizó el modelo de células de McCullch-Pitts añadiéndole aprendizaje y llamando a este modelo el Perceptrón, recogido en su libro *Principles of Neurodynamics (1962).* Este fue sin lugar a duda uno de los avances más importantes en la historia de las redes neurales artificiales.

Dos años después Bernard Widrow diseñó una red muy similar al perceptrón, denominada ADAptative LINear Element, o ADALINE. El adaline de dos niveles era muy similar al perceptrón, a diferencia de que nos daba una función del error en lugar de una salida booleana como en el caso del perceptrón.

A pesar de los brillantes inicios, el interés decayó drásticamente cuando en 1969 Marvin Minsky y Seymour Papert demostraron en su libro *Perceptrons (1969)* las grandes limitaciones teóricas que tenían las redes neuronales, ya que no eran capaces de aprender funciones no linealmente separables. A partir de este trabajo, muchos autores dejaron de lado este campo de la investigación. No obstante, otros autores continuaron con ello, obteniendo grandes resultados, entre los que podemos destacar el asociador linear construido por James Anderson y su extensión conocida como *"Brain-state-in-a-box"*, o diversas redes autoasociativas especializadas en la detección de clusters de ejemplos y la creación de un paradigma de red neural artificial multicapa para visión por Kunihiko Fukushima, recogidas en el libro *Cognition (1975)* y ampliadas posteriormente en el libreo *Neocognition (1980).* Teuvo Kohonen también aportó una gran obra a las redes neuronales. En 1977 Kohonen y Ruhonen extendieron el modelo de memoria asociativa lineal a otro modelo que buscaba las óptimas entre vectores linealmente dependientes, y lo llamó asociador óptimo de memoria lineal (OLAM). Posteriormente desarrolló el LVQ (Linear Vector Quantization) como sistema de aprendizaje competitivo.

En 1982 coincidieron numerosos eventos que hicieron resurgir el interés por las redes neuronales. John Hopfield presentó su trabajo sobre redes neuronales a la Academia Nacional de las Ciencias describiendo con claridad una variante al asociador lineal. Introdujo una función de energía en sus estudios sobre sistemas de ecuaciones no lineales. Esta red es conocida como la "Red de Hopfield". Por su parte, Feldman y Ballard perfeccionaron en 1982 los modelos de redes conexionistas, generalizando sus aplicaciones.
Otro avance significativo tuvo lugar con la formulación de una nueva regla de aprendizaje supervisado, la regla delta generalizada, creada por Werbos, Parker y LeCunn, y popularizada por Rumelhart y McClelland. Así mismo, el algoritmo creado por Rumelhart en 1986 en redes neurales con aprendizaje supervisado conocido como "Backpropagation" ofreció una solución importante para la construcción de redes neuronales más complejas.
En los últimos años, las investigaciones se han centrado en la combinación de ambos paradigmas de aprendizaje, con el objetivo de conseguir sinergias entre la capacidad de procesamiento y aproximación de las redes neuronales artificiales, que pueden llegar a soluciones eficaces a una gran velocidad y con poca información de partida. (Florez López, Fernández Fernández 2008) (Isasi Viñuela, Galván León 2004)

2.3. Modelo computacional

La neurona artificial o autómata es un elemento que posee un estado interno, llamado nivel de activación, y que recibe señales para poder cambiar de estado. Un ejemplo básico podría ser el de encendido/apagado. Una neurona puede valer 0 cuando esté en modo apagado y 1 cuando esté en modo encendido. Sin embargo, se puede utilizar más de un único bit para indicar el estado de la neurona. Por ejemplo, si tuviéramos 8 bits podría tomar de 0 a 255 para mostrar distintos valores en una escala de gris.

Las neuronas poseen una función que les permite cambiar de nivel de activación en función de las señales que van recibiendo. Estas señales pueden provenir tanto del exterior como de otras neuronas. Para calcular el estado de activación se ha de calcular primero la entrada total a la célula. Este valor se calcula como la suma de todas las entradas ponderadas por ciertos valores.

El sumatorio, que corresponde al cuerpo de la neurona, suma todas las entradas ponderadas algebraicamente produciendo una salida, E, así:

$$E = x_1 w_1 + x_2 w_2 + \cdots + x_n w_n \tag{2.1}$$

Esto se puede escribir de forma vectorial como:

$$E = X^T W \tag{2.2}$$

Esta señal E será procesada además por una función denominada función de activación o de salida F. Dependiendo de la función, habrá diferentes tipos de autómatas.

En la figura 2.2 se puede observar las entradas, ponderadas por unos pesos y que dependiendo de un umbral toma un valor u otro (abajo) además de la parte que correspondería a cada parte biológica de la neurona humana (arriba).

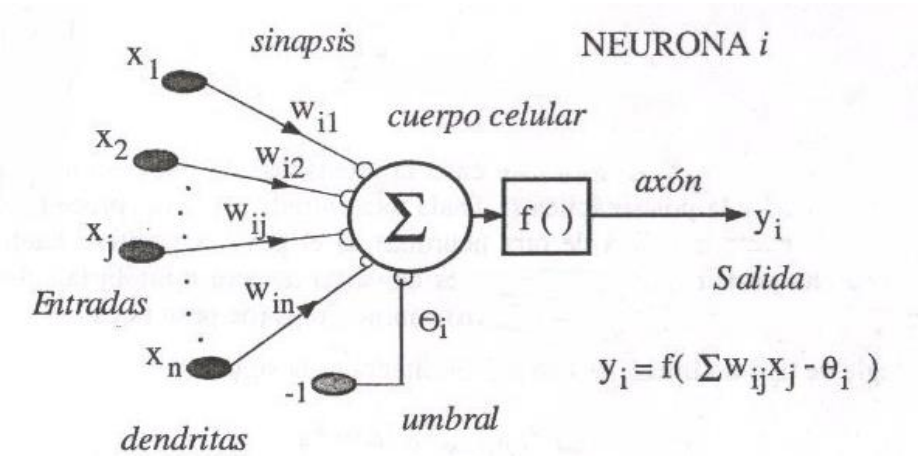


Figura 2.2. Comparación Neuronas biológicas – artificiales (Limoncello 2015)

En la figura 2.3 se puede observar como las entradas se interconectan a otras neuronas que actuarían como el cuerpo de la neurona, éstas podrían conectarse a otras, etc. Hasta llegar a una última capa de salida. Ésta es la estructura básica de una red multicapa. El esquema del funcionamiento de una red de neuronas como el descrito en la figura 2.3 puede escribirse mediante la ecuación:

$$S = F(F(X * W_1) * W_2) \tag{2.3}$$

Donde S es la salida, F son las funciones de activación de las neuronas de cada capa, W_1 son los pesos de la primera capa y W_2 los pesos de la segunda capa. (Isasi Viñuela, Galván León 2004)

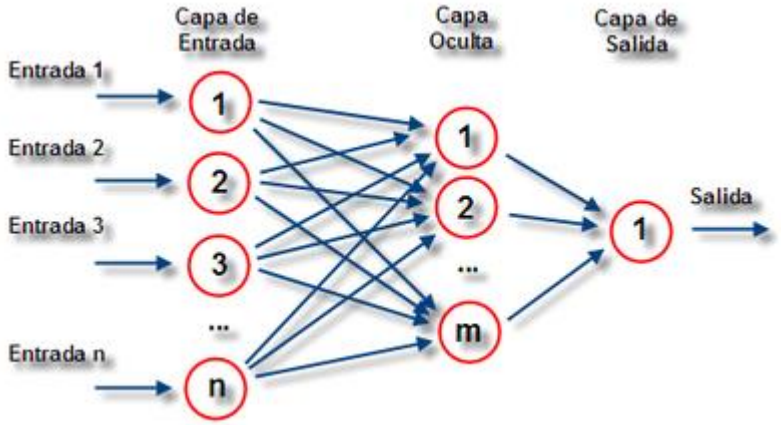


Figura 2.3. Estructura de la red interna(Limoncello 2015)

2.4. Aprendizaje

La parte más importante de una red de neuronas es el aprendizaje. Dependiendo del esquema de aprendizaje una red de neuronas podrá resolver unos problemas u otros. La capacidad de la red estará ligada al tipo de ejemplos del que dispone el proceso de aprendizaje. Este aprendizaje debe ser tanto significativo, es decir, que debe haber un número suficiente de ejemplos, como representativo, es decir, que los componentes deben ser variados.

El proceso general de aprendizaje consiste en ir introduciendo paulatinamente todos los ejemplos del conjunto de aprendizaje, y modificar los pesos de las conexiones siguiendo un determinado esquema. Una vez hecho todo esto, se comprueba si se ha cumplido cierto criterio de convergencia, y en caso de no ser así, se repite el proceso. El criterio de parada depende del tipo de red utilizado. Puede haber distintos criterios de parada, los tres más destacados son los siguientes: a través de un número fijo de ciclos, cuando el error con el esquema de aprendizaje sea inferior a cierta tolerancia o cuando la modificación de los pesos sea irrelevante.

Existen tres tipos de esquemas de aprendizaje:

- Aprendizaje supervisado: Los datos del conjunto de aprendizaje tienen dos tipos de atributos, los datos propiamente dichos y cierta información relativa a la solución del problema. La manera más habitual de modificar los pesos de las conexiones es a través de una comparación entre la salida obtenida y un patrón de datos que ya tenemos. La diferencia entra ambas influirá en cómo se modifican los pesos. Si la diferencia es grande, se modificarán mucho, si no, se modificarán poco.
- Aprendizaje no supervisado: En este caso los datos del conjunto de aprendizaje sólo tienen información sobre los ejemplos, y no hay nada que permita guiar el proceso de aprendizaje. La red irá modificando los pesos a partir de información interna.
- Aprendizaje por refuerzo: Es una variante del aprendizaje supervisado en el que no se dispone de información concreta del error cometido por la red para cada ejemplo de aprendizaje, sino que simplemente determina si la salida es o no la correcta.

Para poder determinar si la red produce salidas adecuadas, el conjunto de aprendizaje se divide en dos grupos, el conjunto de entrenamiento y el conjunto de validación. Con el conjunto de entrenamiento se irá entrenando a nuestra red hasta que se cumpla el criterio de convergencia. El conjunto de validación servirá para comprobar si, tras haber sido entrenada, la red obtiene la salida que se espera. Para ello, el conjunto de validación debe ser independiente del de aprendizaje y debe cumplir las propiedades de un conjunto de entrenamiento. (Isasi Viñuela, Galván León 2004) (Sanchez Camperos, Alanís García 2006)

3. Primeros modelos computacionales

El primer ejemplo que puede considerarse como una red de neuronas artificial son las células de McCullch-Pitts, propuesto en 1943. En él modelizaban la estructura de la neurona como un dispositivo con dos estados, encendido (1) y apagado (0). Si se superaba cierto umbral tras la multiplicación de los valores de entrada por sus respectivos pesos, valía 1, si no, valía 0. Con este primer modelo se podían emular funciones lógicas como la función NOT, la función AND y la función OR. En la función NOT era necesario un único peso de valor -1 y un umbral de valor -1. Si la entrada a la célula es cero, la salida será 0 * (– 1) = 0. Como es mayor que -1, la salida será 1. Para una entrada con valor uno la salida será 1*(-1) = -1, que al no ser mayor que el umbral -1 producirá una salida 0. Para la función AND el valor del umbral es 1, y el de las dos conexiones también lo es. Para la función OR el valor del umbral pasa a ser 0 mientras que el resto de valores de los pesos de las entradas se mantiene. (Florez López, Fernández Fernández 2008)

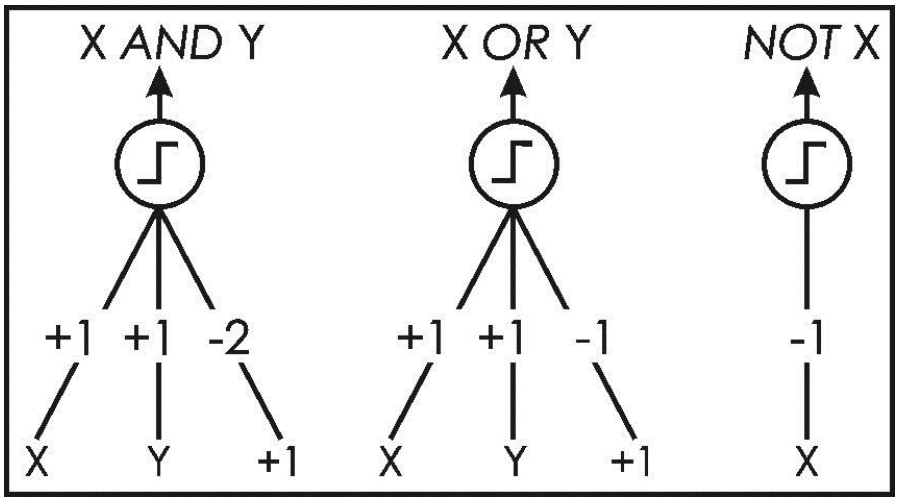


Figura 3.1. Funciones con el modelo McCulloch-Pitts(Florez López, Fernández Fernández 2008)

3.1. Perceptrón

El perceptrón apareció en los años 50 con la aparición de los primeros estudios sobre las redes de neuronas artificiales. Se trata de una arquitectura monocapa en la que hay un conjunto de células de entrada, tantas como sea necesario, y una o varias células de salida. Cada célula de entrada está conectada con todas las células de salida. En la siguiente imagen se puede observar la arquitectura de un perceptrón simple de dos entradas y una salida.

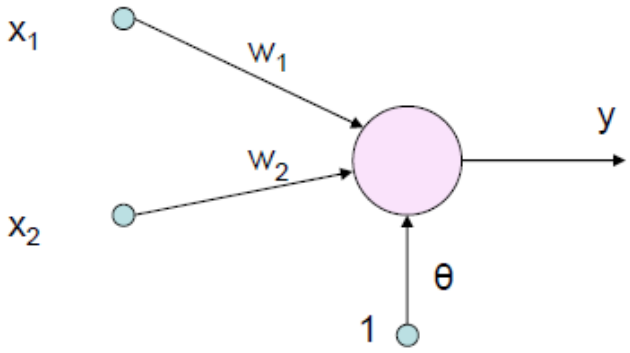


Figura 3.2. Arquitectura del perceptrón simple (Laboratorio Informática UC3M 2015)

El umbral θes un parámetro adicional que se utiliza como factor de comparación para producir la salida, y habrá tantos como células de salida haya en la red. En este esquema, la salida de la red se obtiene de la siguiente forma:

$$y' = \sum_{i=1}^{n} w_i x_i \tag{3.1}$$

La salida definitiva se obtiene al aplicarle la función de salida al nivel de activación de la célula, es decir:

$$y = F(y', \theta) \tag{3.2}$$

Donde *y* será 1 si *y'*> θ y será 0 en caso contrario.

La función F de salida produce una salida binaria, por lo tanto es un diferenciador en dos categorías, que podemos llamar A y B. En el caso de dos dimensiones, la ecuación anterior se transforma en:

$$w_1 x_1 + w_2 x_2 + \theta = 0 \tag{3.3}$$

Que es la ecuación de la recta con pendiente $-\frac{w_1}{w_2}$ y cuyo corte con la abscisa en el en x_1 igual a 0 pasa por $-\frac{\theta}{w_2}$.

Haciendo una descripción gráfica en dos dimensiones, obtendríamos la recta y a cada lado cada una de las dos categorías.

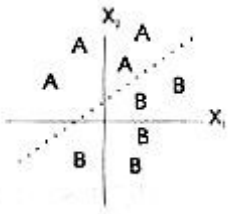


Figura 3.3. Separación en dos categorías mediante un perceptrón. (Isasi Viñuela, Galván León 2004)

Si lo generalizásemos a varias dimensiones:

$$\forall a \in A: w_1 a_1 + \cdots + w_n a_n + \theta > 0$$
$$\forall b \in B: w_1 b_1 + \cdots + w_n b_n + \theta < 0 \tag{3.4}$$

El proceso de aprendizaje consiste en la introducción de un patrón de los del conjunto de aprendizaje perteneciente a una clase. Si la salida es correcta, no se realizará ninguna acción. Si la salida es incorrecta, se modificarán los pesos. (Isasi Viñuela, Galván León 2004)

3.1.1. Función lógica AND con perceptrón

Se propondrá a continuación un ejemplo para una mejor comprensión del perceptrón simple. Se va a desarrollar la función lógica AND.

Se definirán las siguientes entradas y valores:

X1	X2	AND
-1	-1	-1(A)

+1	-1	-1(A)
-1	+1	-1(A)
+1	+1	+1(B)

La iniciación de la red es aleatoria, inicialicemos pues con los pesos a 1 y el umbral a 0,5.

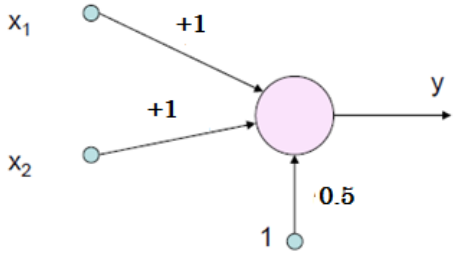


Figura 3.4. Arquitectura del perceptrón simple para ejemplo AND (Laboratorio Informática UC3M 2015)

Con esta inicialización la recta discriminante será $x_1+x_2+0{,}5 =0$. Lo podemos observar en la siguiente imagen:

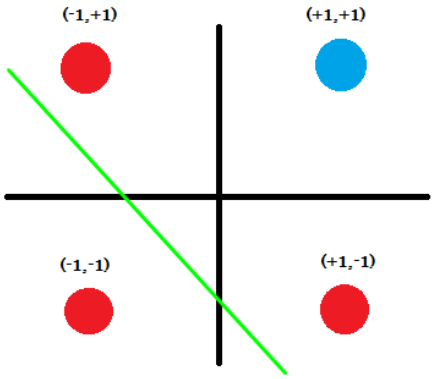


Figura 3.5. Recta determinada por el perceptrón tras la inicialización (Elaboración propia)

Posteriormente se introducen los patrones de entrenamiento y aprendizaje.

Patrón	**Salida**	**Aprendizaje**
(-1,-1|-1)	F(-1 + (-1) + 0,5) = f(-1,5) = -1	Bien [-1 = -1]
(+1,-1|-1)	F(1+(-1)+0,5) = f(0,5) = +1	Mal [+1 ≠ -1]

La salida del primer patrón es correcta, por lo tanto no hay variaciones en los pesos. Sin embargo, en la segunda salida la clasificación es errónea, por lo tanto habrá una actualización de los pesos. Esta actualización se realizará tal y como se refleja en la siguiente tabla:

Aprendizaje

Antes	Incremento	Después
W1 = 1	d(x) * x1 = -1 * 1 = -1	W1 = 0
W2 = 1	d(x) * x2 = -1 * -1 = +1	W2 = 2
Θ = 0,5	d(x) = -1	Θ = -0,5

De esta forma se modifican los pesos y se describe la recta $2x_2 - 0{,}5 = 0$, que podemos observar en la siguiente figura.

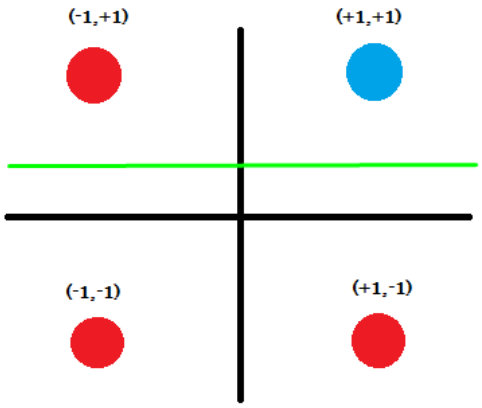


Figura 3.6. Recta determinada por el perceptrón tras la primera iteración (Elaboración propia)

Continuamos introduciendo los patrones:

Patrón	Salida	Aprendizaje
(-1,+1|-1)	F(0+2-0,5) =f(1,5) = +1	Mal [+1≠ -1]

De nuevo, volvemos a la fase de aprendizaje:

Aprendizaje

Antes	Incremento	Después
W1 = 0	d(x) * x1 = -1 * -1 = 1	W1 = 1
W2 = 2	d(x) * x2 = -1 * 1 = -1	W2 = 1
Θ = -0,5	d(x) = -1	Θ = -1,5

Tras este ciclo de aprendizaje la recta quedaría así: $x_1 + x_2 - 1{,}5 = 0$, podemos observar esta recta en la siguiente figura.

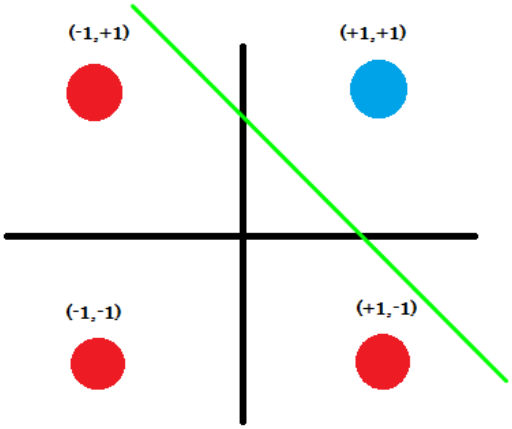


Figura 3.7. Recta determinada por el perceptrón tras la segunda iteración (Elaboración propia)

Visualmente se puede apreciar como ahora la recta clasifica ambas regiones correctamente, pero para asegurarnos haremos una nueva iteración a mayores.

Patrón	Salida	Aprendizaje
(-1,-1|-1)	F(-1 + (-1) -1,5) = f(-3,5) = -1	Bien [-1 = -1]
(+1,-1|-1)	F(1+(-1)-1,5) = f(-1,5) = -1	Bien [-1 = -1]
(-1,+1|-1)	F(-1+1-1,5) =f(-1,5) = -1	Bien [-1 = -1]
(+1,+1|+1)	F(1+1-1,5) = f(0,5) = 1	Bien [+1 = +1]

Y de esta forma quedaría terminado el aprendizaje del perceptrón. (Isasi Viñuela, Galván León 2004)

3.2. Adaline

En el año 1960, Widrow y Hoff propusieron un sistema de aprendizaje en el que sí que se tuviera en cuenta el error producido, de forma que a mayor error la variación de los pesos fuera mayor, y viceversa. A este sistema lo llamaron ADAptive Linear NEuron, o ADALINE. Con una estructura idéntica al perceptrón, su diferencia se basa en la función de salida de cada una de las neuronas. Mientras que en el perceptrón la función de salida debe ser siempre una función en escalón, una salida binaria, en Adaline la función de salida era un clasificador en varios números reales. Al ser salidas binarias, sólo se podían codificar un número discreto de estados, sin embargo, si las salidas fueran número reales, esta red se convertiría en un sistema de resolución de problemas generales.

Aunque la mayor diferencia entre ambos era la función de salida de cada una de las neuronas, existían otro tipo de diferencias que hacían de Adaline una red de neuronas más sofisticada que el perceptrón simple. Mientras que en el perceptrón la diferencia entre entrada y salida es 0 si ambas pertenecen a la misma clase y ±1 si pertenecen a clases distintas, en Adaline se calcula la diferencia real entre la salida obtenida y la deseada, esto hace que Adaline pueda mostrar cuánto se ha equivocado nuestra red, mientras que el perceptrón simple únicamente nos decía si se

había equivocado o no. Además, en Adaline existe una razón de aprendizaje (γ) para regular cuanto va a afectar cada equivocación a la modificación de los pesos en el aprendizaje. (Isasi Viñuela, Galván León 2004) (Martín del Brío, Sanz Molina 2006)

4. Perceptrón Multicapa

El perceptrón multicapa es un modelo de red neuronal surgido en los años 80 con el objetivo de solucionar el problema detectado por el perceptrón simple, esto es, la imposibilidad de aprender clases de funciones no linealmente separables (como la función XOR). Se ha demostrado que el perceptrón multicapa es un aproximador universal de funciones, de forma que puede aproximar cualquier función continua en R^n.

Este perceptrón se conoce como un modelo computacional de red de neuronas con conexiones hacia delante, caracterizado por tener salidas disjuntas relacionadas entre sí, es decir, que la salida de una neurona es la entrada de la siguiente. Este tipo de relaciones se organizan en capas, de forma que la salida de una neurona de una capa posterior no pueda ser la entrada de una neurona de una capa anterior, ya que eso serían conexiones hacia atrás y en el perceptrón las conexiones son únicamente hacia delante.

La arquitectura del perceptrón multicapa se caracteriza porque tiene las neuronas agrupadas en distintos niveles o capas. Cada una de las capas está formada por un conjunto de neuronas y se pueden distinguir tres tipos de capas, capa de entrada, capa de salida y capas ocultas.

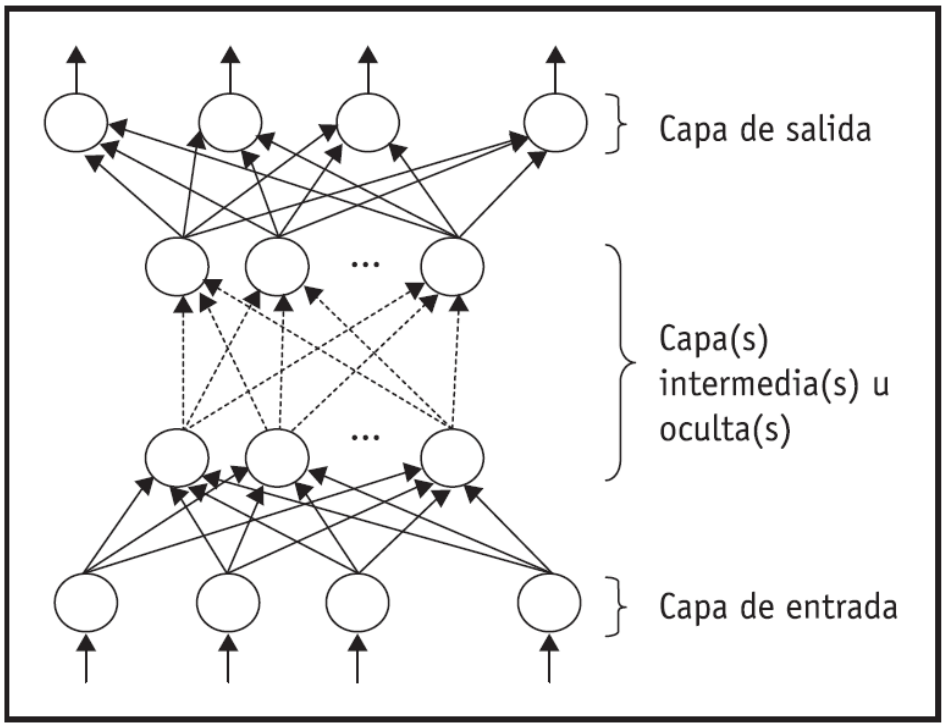


Figura 4.1. Arquitectura del perceptrón multicapa (Florez López, Fernández Fernández 2008)

Las neuronas de la capa de entrada sirven para recibir las señales o patrones de aprendizaje, por lo tanto no actúan realmente como neuronas en sí. Las neuronas de la capa de salida son las que se comunican con el exterior, y la salida de éstas son las que se muestra como salida del perceptrón multicapa. Podemos observar de nuevo en la figura 4.1 cómo las conexiones entre

neuronas están siempre dirigidas hacia delante. A este tipo de redes se les conoce como redes hacia delante o redes *feedforward*.

Normalmente todas las neuronas de la capa anterior están conectadas a las neuronas de la capa posterior. Se dice entonces que existe conectividad total, o que la red está totalmente conectada.

En el perceptrón multicapa, al igual que en el perceptrón, existe una fase de propagación de los patrones de entrada hacia delante, y una fase de aprendizaje, en el que lo datos obtenidos a la salida del perceptrón se van propagando hacia atrás para observar el error de cada una de las neuronas y actualizar los pesos de éstas. Este algoritmo se denomina algoritmo de retropropagación y hablaremos de él más adelante con detalle. (Isasi Viñuela, Galván León 2004) (Martín del Brío, Sanz Molina 2006)

4.1. Propagación de los patrones de salida

El perceptrón multicapa define una relación entre las variables de entrada y las variables de salida de la red. Esta relación se obtiene propagando las entradas a través de las diferentes capas de la red hasta llegar a la capa de salida. Cada una de las neuronas actúa por separado, recibiendo una entrada, realizando la función determinada y propagando una salida a la siguiente neurona.

Sea un perceptrón multicapa con C capas. Denominaremos $W^C = w_{ij}^c$ a la matriz de pesos asociada a las conexiones de la capa c a la capa c+1 donde w_{ij}^c representa el peso de la conexión de la neurona i de la capa c a la neurona j de la capa c+1. Denominaremos también $U^C = u_i^c$ al vector de umbrales de las neuronas de la capa c. Se denomina a_i^c a la activación o entrada de la neurona i de la capa c. Las activaciones de las neuronas se calcularán de forma distinta en función de la capa en la que nos encontremos.

- Activación de las neuronas de la capa de entrada (a_i^1): Estas activaciones corresponderán con las entradas del perceptrón.
- Activación de las neuronas de la capa oculta (a_i^c): Estas activaciones procederán de las salidas de las neuronas de la capa anterior que están conectadas a la neurona i de la capa c. esta activación se calcula como la suma de los productos de las activaciones que reciben las neuronas de las capas anteriores multiplicadas por su respectivo peso, añadiéndoles el umbral y aplicando la función de activación, es decir:

$$a_i^c = f(\sum_{j=1}^{n_{c-1}} w_{ji}^{c-1} a_j^{c-1} + u_i^c) \text{ para i = 1,2…} n_c \text{ y c = 2,3…C-1} \quad (4.1)$$

- Activación de las neuronas de la capa de salida (a_i^C): Al igual que en el caso anterior, viene dada por la misma ecuación, a diferencia de que ahora la salida de las neuronas corresponderá con la salida de la red.

$$y_i = f(\sum_{j=1}^{n_{C-1}} w_{ji}^{C-1} a_j^{C-1} + u_i^C) \text{ para i = 1,2…} n_c \quad (4.2)$$

La función f de la que se ha hablado anteriormente es la función de activación. Para el perceptrón multicapa hay dos funciones de activación características, que suelen ser las más usadas, la función tangente hiperbólica y la función sigmoidal. Ambas funciones son muy parecidas y se puede pasar de una a otra de una forma sencilla. La principal diferencia entre ellas es el nivel de saturación inferior, que para la función sigmoidal es 0 y para la función tangente hiperbólica es -1. Para ambas, el nivel de saturación superior es 1.

Las expresiones de las funciones son las siguientes:

- Función sigmoidal:

$$f_1(x) = \frac{1}{1 + e^{-x}} \tag{4.3}$$

- Función tangente hiperbólica:

$$f_2(x) = \frac{1 - e^{-x}}{1 + e^{-x}} \tag{4.4}$$

Y las gráficas de ambas son las siguientes:

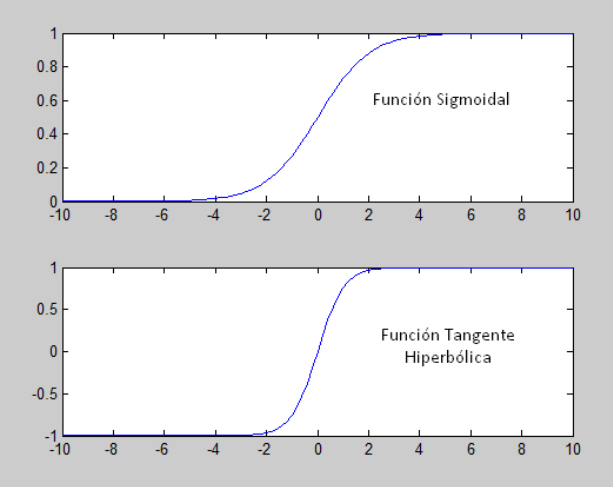


Figura 4.2. Funciones de activación del perceptrón multicapa (Elaboración propia)

Ambas funciones están relacionadas mediante la expresión $f_2(x) = 2f_1(x) - 1$, por lo que la utilización de una u otra se elige en función de cual sea más compatible con el tipo de patrón que se va a utilizar.

Normalmente la función de activación es la misma para todas las neuronas de la red, sin embargo, en las neuronas de la capa de salida es posible que se utilicen otros dos tipos de funciones distintas. Estas funciones son la función escalón y la función identidad. (Isasi Viñuela, Galván León 2004)

4.2. Algoritmo de retropropagación

Como se ha explicado anteriormente, el proceso de aprendizaje consistirá en propagar hacia atrás cierta información desde la salida desde perceptrón hasta su entrada. Esta información consistirá en el error entre la salida deseada y la salida obtenida. Pero el algoritmo es bastante más complejo, ya que en cada capa deberemos eliminar todas las variaciones que la salida haya sufrido durante sus capas posteriores. De este modo tendremos la activación exacta para cada neurona.

El objetivo del aprendizaje será el de reducir al mínimo el error. Hay diversos errores que se pueden utilizar, en el caso del perceptrón multicapa el error a minimizar será un error similar al error cuadrático medio, este error se definirá como:

$$e(n) = \frac{1}{2}\sum_{i=1}^{n_c}(s_i(n) - y_i(n))^2 \quad (4.5)$$

Siendo n cada una de las iteraciones del aprendizaje.

El error obtenido e(n) es el error de un solo patrón. En el caso de que haya que entrenar con varios patrones, el error a minimizar será el error medio de éste:

$$E = \frac{1}{N}\sum_{n=1}^{N} e(n) \quad (4.6)$$

Con lo cual, el aprendizaje del perceptrón multicapa es equivalente a encontrar un mínimo en la función del error.

Aunque el aprendizaje deba hacerse minimizando el error total, el procedimiento más utilizado es el basado en métodos del gradiente estocástico, que consisten en minimizar los errores para cada patrón e(n). Aplicando el método del gradiente estocástico obtenemos que la actualización de los pesos deberá realizarse de acuerdo a:

$$w(n) = w(n-1) - \alpha\frac{\partial e(n)}{\partial w} \quad (4.7)$$

Donde e(n) es el error para el patrón n y α es la razón o tasa de aprendizaje.

Aplicando este algoritmo a todas las neuronas obtenemos el conocido algoritmo de retropropagación o regla delta generalizada. Pasemos ahora a obtener esta regla.

Dentro de la regla delta, es importante distinguir dos categorías esenciales, la última capa y las capas intermedias. Comenzaremos por la última capa ya que es la más sencilla y la que da pie a poder analizar el comportamiento de la regla delta en las capas intermedias.

Pesos y umbrales de la capa oculta C-1 a la capa de salida

Siendo w_{ij}^{c} el peso de la conexión de la neurona i de la capa c a la neurona j de la capa c+1, sabiendo que tenemos C capas, la ecuación anterior quedaría como:

$$w_{ji}^{C-1}(n) = w_{ji}^{C-1}(n-1) - \alpha\frac{\partial e(n)}{\partial w_{ji}^{C-1}} \quad (4.8)$$

Es decir, que para actualizar dicho peso es necesario obtener la derivada del error respecto al peso en dicho punto. De acuerdo con la expresión del error, que las salidas deseadas son constantes y que el peso w_{ij}^{C-1} sólo afecta a la neurona de salida i, obtenemos:

$$\frac{\partial e(n)}{\partial w_{ji}^{C-1}} = -(s_i(n) - y_i(n))\frac{\partial y_i(n)}{\partial w_{ji}^{C-1}} \tag{4.9}$$

Ahora nos encontramos con que tenemos que obtener la derivada de la salida respecto al peso w_{ij}^{C-1}. Tomando la ecuación de $y_i(n)$ y derivándola respecto del peso correspondiente, sabiendo que de los términos del sumatorio el único en el que interviene el peso w_{ji}^{C-1} es $w_{ji}^{C-1}a_j^{C-1}$, y por tanto el único cuya derivada es distinta de cero se obtiene:

$$\frac{\partial y_i(n)}{\partial w_{ji}^{C-1}} = f'(\sum_{j=1}^{n_{C-1}} w_{ji}^{C-1}a_j^{C-1} + u_i^C)a_j^{C-1}(n) \tag{4.10}$$

Definimos el término δ asociado a la neurona i de la capa de salida como:

$$\delta_i^C = -(s_i(n) - y_i(n))f'(\sum_{j=1}^{n_{C-1}} w_{ji}^{C-1}a_j^{C-1} + u_i^C) \tag{4.11}$$

Con lo que quedaría la ecuación anterior como:

$$\frac{\partial e(n)}{\partial w_{ji}^{C-1}} = \delta_i^C a_j^{C-1}(n) \tag{4.12}$$

Y finalmente reemplazando en la actualización de los pesos obtendríamos:

$$w_{ji}^{C-1}(n) = w_{ji}^{C-1}(n-1) - \alpha\delta_i^C a_j^{C-1}(n) \tag{4.13}$$

Así mismo, para la actualización de los umbrales habría que seguir el mismo procedimiento, de forma que éstos quedarían de la siguiente forma:

$$u_i^{C-1}(n) = u_i^{C-1}(n-1) - \alpha\delta_i^C(n) \tag{4.14}$$

Es decir, igual que para los pesos pero como si la activación fuese igual a 1. Este es un detalle importante para la eficiencia de cálculo en cuanto a computación se refiere, ya que podremos agrupar los umbrales con los pesos, algo de lo que hablaremos posteriormente.

Pesos y umbrales de la capa oculta c a la capa oculta c+1

Para una mayor claridad a la hora de comprender el procedimiento, se van a tomar los pesos y umbrales de la capa C-2 a la capa C-1. Será w_{kj}^{C-2} el peso de la conexión entre la neurona k de la capa C-2 a la neurona j de la capa C-1. Seguimos deseando minimizar el error total, que de

nuevo basta con minimizar el error para cada uno de los patrones. Utilizamos el método del descenso del gradiente, de modo que la ley viene dada por:

$$w_{kj}^{C-2}(n) = w_{kj}^{C-2}(n-1) - \alpha \frac{\partial e(n)}{\partial w_{kj}^{C-2}} \tag{4.15}$$

En este caso, a diferencia de en el anterior, el peso w_{kj}^{C-2} **SI** que influye en todas las salidas de la red, por lo que la derivada del error viene dada por la suma de todas las derivadas en las que este peso tiene influencia, es decir:

$$\frac{\partial e(n)}{\partial w_{kj}^{C-2}} = -\sum_{i=1}^{n_C} (s_i(n) - y_i(n)) \frac{\partial y_i(n)}{\partial w_{kj}^{C-2}} \tag{4.16}$$

Para calcular la derivada de la salida respecto al peso w_{kj}^{C-2} es necesario tener en cuenta que este peso influye en la neurona j de la capa C-1, pero no en el resto, por lo tanto la derivada del resto será 0 en todas las neuronas excepto en la neurona j. Por lo tanto obtenemos:

$$\frac{\partial y_i(n)}{\partial w_{kj}^{C-2}} = f'(\sum_{j=1}^{n_{C-1}} w_{ji}^{C-1} a_j^{C-1} + u_i^C) w_{ji}^{C-1} \frac{\partial a_j^{C-1}}{\partial w_{kj}^{C-2}} \tag{4.17}$$

Sustituyendo este valor en la ecuación anterior y de acuerdo con el valor de δ obtenido en el punto anterior, obtenemos:

$$\frac{\partial e(n)}{\partial w_{kj}^{C-2}} = \sum_{i=1}^{n_C} \delta_i^C(n) w_{ji}^{C-1} \frac{\partial a_j^{C-1}}{\partial w_{kj}^{C-2}} \tag{4.18}$$

Finalmente, para obtener la ley de aprendizaje basta con derivar la función de activación respecto del peso. Aplicamos la regla de la cadena en las funciones del apartado de propagación de los patrones de salida y obtenemos:

$$\frac{\partial a_j^{C-1}}{\partial w_{kj}^{C-2}} = f'(\sum_{k=1}^{n_{C-2}} w_{kj}^{C-2} a_k^{C-2} + u_j^{C-1}) a_k^{C-2}(n) \tag{4.19}$$

Finalmente definimos el valor δ para las neuronas de la capa C-1 como:

$$\delta_j^{C-1}(n) = f'(\sum_{k=1}^{n_{C-2}} w_{kj}^{C-2} a_k^{C-2} + u_j^{C-1}) \sum_{i=1}^{n_C} \delta_i^C(n) w_{ji}^{C-1} \tag{4.20}$$

Y sustituyendo la nueva $\delta_j^{C-1}(n)$ en la ecuación del error obtenemos:

$$\frac{\partial e(n)}{\partial w_{kj}^{C-2}} = \delta_j^{C-1}(n)\, a_k^{C-2}(n) \tag{4.21}$$

que para actualizar los pesos de la capa C-2, obtendríamos de la siguiente forma:

$$w_{kj}^{C-2}(n) = w_{kj}^{C-2}(n-1) - \alpha\delta_j^{C-1}(n)\, a_k^{C-2}(n) \tag{4.22}$$

Finalmente, extendemos todo lo desarrollado para una capa oculta c cualquiera, de modo que la modificación de pesos quedaría como:

$$w_{kj}^{c}(n) = w_{kj}^{c}(n-1) - \alpha\delta_j^{c+1}(n)\, a_k^{c}(n) \tag{4.23}$$

donde $\delta_j^{c+1}(n)$ es:

$$\delta_j^{c+1}(n) = f'\Big(\sum_{k=1}^{n_c} w_{kj}^{c} a_k^{c} + u_j^{c}\Big) \sum_{i=1}^{n_{c+1}} \delta_i^{c+2}(n) w_{ji}^{c} \tag{4.24}$$

Por último, actualizamos los umbrales de la misma forma que los pesos, y al igual que en la última capa, se puede desarrollar la actualización de éstos como si la activación fuese constante e igual a 1.

$$u_j^{c+1}(n) = u_j^{c+1}(n-1) - \alpha\delta_j^{c+1}(n) \tag{4.25}$$

Como conclusión, cabe destacar la influencia de una delta de la capa posterior en la delta de la capa anterior. Esto hace que el error, y por consiguiente el delta, se vaya retropropagando, o propagando hacia atrás, a través de todas las neuronas.

Derivada de la función de activación

El cálculo de los valores de δ requiere conocer la derivada de la función de activación. Recordamos que la función de activación puede tomar dos formas, una función sigmoidal y una función tangente hiperbólica.

- Derivada de la función sigmoidal

Derivando la función sigmoidal obtenemos:

$$f_1'(x) = \frac{1}{(1+e^{-x})^2}(e^{-x}) = \frac{1}{(1+e^{-x})}\,\frac{e^{-x}}{(1+e^{-x})} \tag{4.26}$$

Es decir:

$$f_1'(x) = f_1(x)(1 - f_1(x)) \tag{4.27}$$

Como consecuencia, los valores δ asociados quedarían como:

$$\delta_i^C = -(s_i(n) - y_i(n))y_i(n)(1 - y_i(n)) \tag{4.28}$$

para las neuronas de salida y

$$\delta_j^{c+1} = a_j^c(n)(1 - a_j^c(n)) \sum_{i=1}^{n_{c+1}} \delta_i^{c+2}(n) w_{ji}^c \tag{4.29}$$

para el resto de neuronas de la red.

- Derivada de la función tangente hiperbólica

Teniendo en cuenta que $f_2(x) = 2f_1(x) - 1$, la derivada de la función $f_2(x)$ es:

$$f_2'(x) = 2f_1(x)(1 - f_1(x)) \tag{4.30}$$

En el caso de las neuronas de salida, se ha comentado que a veces utilizan como función de activación la función escalón o la función identidad. En este caso, la derivada es igual a 1 y los valores δ quedan determinados por:

$$\delta_i^C = -(s_i(n) - y_i(n)) \tag{4.31}$$

(Isasi Viñuela, Galván León 2004) (Martín del Brío, Sanz Molina 2006) (Sanchez Camperos, Alanís García 2006) (Flórez López, Fernández Fernández, 2008)

4.2.1. Razón de aprendizaje y momento de aprendizaje

El parámetro α o razón de aprendizaje determina la velocidad a la que van a cambiar los pesos de las conexiones entre las neuronas. Se encuentra entre el rango [0,1] y si se acerca más a cero, los pesos cambiarán poco y se irá acercando lentamente hasta la convergencia. Si se acerca más a uno, irá convergiendo rápidamente al principio pero es posible que oscile entre el valor del peso correcto al final, ya que cambiará demasiado deprisa. Por ello hay que buscar una razón adecuada.

Por ello se incluye un segundo término, denominado momento. Este término nos pondera cuánto queremos que influya lo que los pesos han cambiado en la iteración anterior. De esta forma, si los pesos han cambiado mucho, significa que estamos lejos y eso hará que avancemos aún más rápido, sin embargo, si en la iteración anterior los pesos han cambiado poco, el

momento hará que las variaciones no sean tan bruscas como si sólo tuviéramos la razón α. Al momento se le suele determinar con el término η.

Si añadimos el momento a nuestra ecuación, la actualización (tanto de pesos como de umbrales) quedaría de la siguiente forma:

$$w(n) = w(n-1) - \alpha \frac{\partial e(n)}{\partial w} + \eta \Delta w(n-1) \tag{4.32}$$

donde $\Delta w(n-1) = w(n-1) - w(n-2)$ es el incremento que sufrió el parámetro w en la iteración anterior. (Neural networks and Deep learning 2015)

4.2.2. Ejemplo. Regla delta generalizada para un perceptrón con dos neuronas de entrada, dos ocultas y una salida

Vamos a poner un ejemplo para que los conceptos teóricos sean más sencillos. Tenemos un perceptrón con dos entradas, por lo tanto en la capa de entrada tendremos dos neuronas, dos neuronas en la capa oculta, y una neurona en la capa de salida, por lo que tenemos una salida. La imagen sería así:

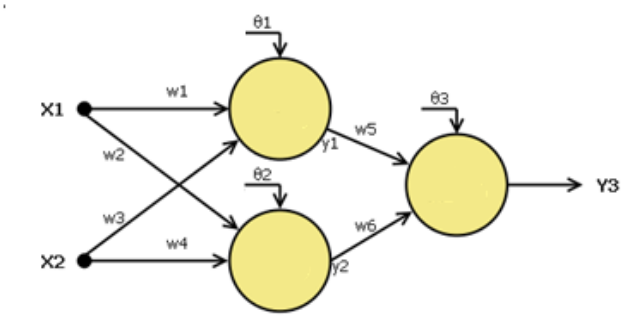


Figura 3.3. Perceptrón de dos capas(Censor Cósmico 2015)

Si observamos la expresión de actualización de los pesos de la última capa:

$$w_{ji}^{C-1}(n) = w_{ji}^{C-1}(n-1) - \alpha \delta_i^C a_j^{C-1}(n) \tag{4.33}$$

Tendríamos que:

$$w_5(n) = w_5^2(n-1) - \alpha \delta^3 y_1(n) \tag{4.34}$$

$$w_6(n) = w_6^2(n-1) - \alpha \delta^3 y_2(n) \tag{4.35}$$

Y para el umbral:

$$\theta_3(n) = \theta_3(n-1) - \alpha \delta^3(n) \tag{4.36}$$

Si observamos la capa oculta, la fórmula para la actualización de pesos obtenida en la teoría es:

$$w_{kj}^{c}(n) = w_{kj}^{c}(n-1) - \alpha\delta_{j}^{c+1}(n)\, a_{k}^{c}(n) \tag{4.37}$$

que para los pesos w_1,w_2,w_3 y w_4 es:

$$w_1(n) = w_1(n-1) - \alpha\delta_1^2(n)\, x_1(n) \tag{4.38}$$

$$w_2(n) = w_2(n-1) - \alpha\delta_2^2(n)\, x_1(n) \tag{4.39}$$

$$w_3(n) = w_3(n-1) - \alpha\delta_1^2(n)\, x_2(n) \tag{4.40}$$

$$w_4(n) = w_4(n-1) - \alpha\delta_2^2(n)\, x_2(n) \tag{4.41}$$

Y para los umbrales θ_1 y θ_2:

$$\theta_1(n) = \theta_1(n-1) - \alpha\delta_1^2(n) \tag{4.42}$$

$$\theta_2(n) = \theta_2(n-1) - \alpha\delta_2^2(n) \tag{4.43}$$

Siendo $\delta_1, \delta_2\ y\ \delta_3$:

$$\delta_1 = f'(w_1x_1 + w_3x_2 + \theta_1)w_5\delta_3 \tag{4.44}$$

$$\delta_2 = f'(w_2x_1 + w_4x_2 + \theta_2)w_6\delta_3 \tag{4.45}$$

$$\delta_3 = -(s(n) - y_3(n))f'(w_5y_1 + w_6y_2 + \theta_3) \tag{4.46}$$

(Isasi Viñuela, Galván León 2004)

5. Marco de estudio de las tarjetas gráficas

En los últimos años se han producido grandes avances en el ámbito de la computación. El hardware es mucho más potente y el software es más eficiente, lo que hace que los programas se ejecuten rápidamente. Existe un punto de inflexión en éste ámbito. A principios del siglo XXI se ideó una forma de utilizar las tarjetas gráficas (GPU) de los ordenadores (que eran más potentes a nivel computacional que las CPUs) como módulos de procesamiento de carácter general (GPGPU – General Purpose Graphics Processing Unit). Gracias a este procedimiento se consiguió avanzar de una forma abrumadora y se obtuvo un tremendo éxito a la hora de observar la velocidad de procesamiento de los nuevos programas. Programas que antes tardaban horas en ejecutarse, podían ser ejecutados en cuestión de segundos.

La GPU (Graphics Processing Unit) era el dispositivo encargado de realizar este tipo de operaciones. Mientras que una CPU podía llegar a tener 8 núcleos (o procesadores) con los que trabajar, una GPU común tenía 512 núcleos. Esto es claramente más rápido, computacionalmente hablando, si se puede aprovechar de una forma exitosa. Para ello se idearon nuevos lenguajes de programación ya que los anteriores no servían porque requerían la necesidad de explotar el paralelismo de los 512 núcleos de las tarjetas gráficas. Otros lenguajes dedicados únicamente para las tarjetas gráficas, como las directivas OpenGL o los Shaders Cg tuvieron dificultades a la hora de traspasar para que tuvieran un propósito general, aunque finalmente tuvieran éxito. También surgieron iniciativas en diferentes universidades para que este tipo de cálculos fueran más sencillos. Otro ejemplo son los lenguajes dedicados que surgieron por aquel entonces, como BrookGPU.

OpenCL surgió también como una solución a este tipo de problemas, es una API de bajo nivel para computación heterogénea en GPUs compatibles con el lenguaje CUDA. La ventaja de OpenCL es que con el uso de la API OpenCL, los desarrolladores pueden lanzar kernels de computación usando un subconjunto limitado del lenguaje de programación C en una GPU.

Sin embargo, hasta 2007 no fue cuando NVIDIA desarrolló su CUDA Toolkit para la computación sobre GPGPU. Esto supuso una nueva plataforma de cálculo y un nuevo modelo de programación desarrollado sobre C/C++. Posteriormente se ha ampliado a otros lenguajes como Fortran o Python. (NVIDIA España 2015)

6. Introducción y soporte CUDA

Para la realización del trabajo propuesto va a ser necesaria una forma de poder utilizar las tarjetas gráficas de nuestra computadora a un nivel óptimo. Para ello, se pueden utilizar diferentes herramientas. En este proyecto se utilizará CUDA sobre C++, desarrollado sobre la herramienta Nsight de NVIDIA. Esta herramienta tiene un entorno similar a Eclipse, el programa de compilación de programas básico del sistema operativo Linux. En nuestro caso, será necesario la utilización de un compilador aparte, el NVCC o NVIDIA compiler, que será el encargado de compilar nuestros programas generados en Nsight.

6.1. Lenguaje C++

C++ es un lenguaje de programación. Los programas, para ser ejecutados, deben estar escritos en código máquina, difícilmente entendible por un ser humano. C++ es un lenguaje con el propósito hacer más fácil esta tarea, a modo de un lenguaje de "traducción" entre un humano y una máquina.

C++ es un lenguaje abierto y estandarizado por la ISO desde 1998. Es un lenguaje compilado, es decir, que compila directamente a código máquina, sin escalas intermedias. Es un lenguaje fuertemente tipificado, lo que permite al programador una gran cantidad de opciones siempre y cuando éste sepa lo que está haciendo, es decir, que requiere un gran conocimiento por parte del programador. Permite tanto la declaración de variables de forma manifiesta o inferida, y permite tanto la comprobación dinámica como estática de los tipos de datos. Se puede utilizar en múltiples plataformas (como Eclipse, Nsight, Visual Studio…) y es compatible con C. Tiene una gran cantidad de librerías que apoyan todo el proyecto y mejoran su rendimiento. (CPLUSPLUS 2015)

6.2. Nsight Eclipse Edition

NVIDIA®Nsight™ es una plataforma de desarrollo para la computación heterogénea. Trabaja con una poderosa depuración y herramientas que permiten optimizar al máximo el rendimiento de la CPU y la GPU. Esta herramienta no solo es capaz de optimizar el rendimiento sino que además también ayuda a obtener una mejor comprensión del código - Identificar y analizarlos cuellos de botella y observar el comportamiento de todas las actividades del sistema.

Dentro de NVIDIA Nsight existen dos soportes principales, Visual Studio, para Windows, y Eclipse, para Linux. Como en este proyecto se ha estado utilizando Linux, será la versión sobre Eclipse la que se utilizará en este proyecto. (NVIDIA Developer 2015)

6.3. Eclipse

Eclipse es un programa soportado por Linux compuesto por un conjunto de herramientas de código abierto utilizadas para el desarrollo de IDE (Integrated Development Environment).

Eclipse dispone de un editor de texto con compilador en tiempo real, capaz de detectar fallos y problemas sintácticos mientras se está redactando código. Tiene potentes herramientas de depuración, además de un entorno sencillo y manejable.

Dispone de herramientas adicionales en función de los lenguajes de programación que estemos utilizando, en nuestro caso, como trabajaremos con C++ y CUDA, utilizaremos el C/C++ Development Toolkit (CDT) y el CUDA Toolkit.

El **C/C++ DevelopmentToolkit (CDT)** son una serie de extensiones de C/C++ que amplía el área de trabajo de Eclipse para poder trabajar con C/C++ de una forma más eficiente. Proporciona herramientas como la construcción, la compilación y la depuración el código.

El **CUDA Toolkit** son una serie de extensiones de CUDA que amplía el área de trabajo existente en Eclipse para poder trabajar con dispositivos GPGPU, que usen una o más tarjetas gráficas NVIDIA. Incluye el **NVIDIA NVCC Compiler**, el compilador de NVIDIA basado en código abierto LLVM, que utilizaremos para compilar nuestro código escrito en CUDA, mientras que el código en C++ lo compilaremos con el gcc/g++. (NVIDIA Developer 2015) (Eclipse 2015) (Help Eclipse Luna 2015)

7. Herramienta CUDA

CUDA es el acrónimo de Computed Unified Device Architecture. CUDA es una arquitectura de cálculo en paralelo ideada por NVIDIA. Esta arquitectura aprovecha la potencia de la GPU para

que, trabajando con datos que no tengan que ser obligatoriamente gráficos, ésta sea capaz de acelerar el rendimiento de los programas que deberían ser ejecutados en la CPU.

CUDA también hace referencia al lenguaje de programación utilizado. Es un lenguaje que permite al programador un amplio abanico de posibilidades, sin embargo, requiere de éste que sepa perfectamente lo que está haciendo.

Se ha observado que la diferencia entre el rendimiento computacional de la CPU y la GPU es abrumador.

En la figura 7.1 podemos observar la diferencia tanto en ancho de banda como en número de operaciones por segundo (FLOPs) de la CPU respecto a la GPU a lo largo de los años.

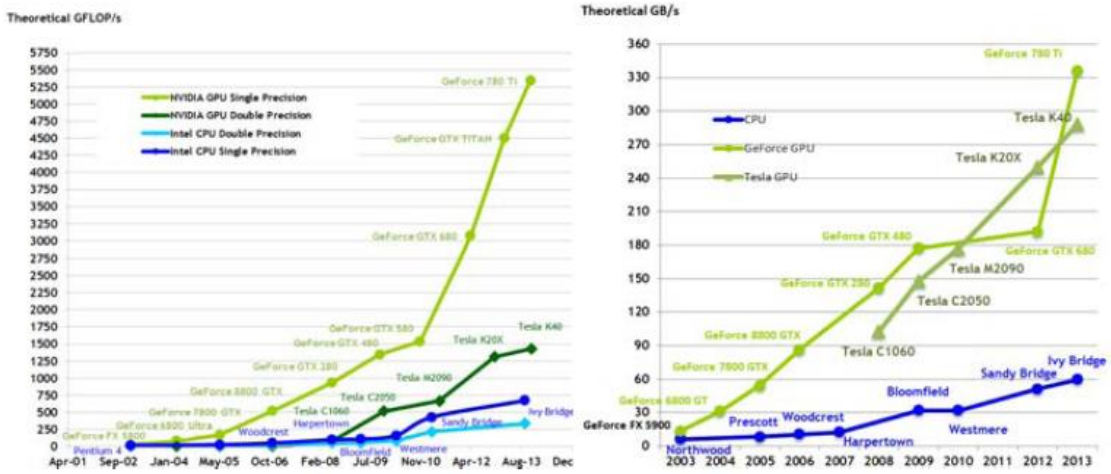


Figura 7.1. Rendimiento computacional y ancho de banda CPU vs GPU (NVIDIA España 2015)

Como podemos observar, desde prácticamente el inicio de las GPU como computadores de propósito general, han destacado sobre las CPU. Mientras que las CPU en precisión simple no eran capaces de llegar a más de 750 GFLOP/s, la GPU de NVIDIA era capaz de llegar a los 5.5 TFLOP/s. Estamos hablando de una diferencia de un orden de magnitud, algo realmente importante si estamos buscando la eficiencia en nuestro trabajo. Si observamos en doble precisión, la GPU, con 1500 GFLOP/s, de nuevo sobrepasaba a la CPU, que sólo alcanzaba los 300 GFLOP/s.

Si nos fijásemos en el ancho de banda, de nuevo las GPUs ganan con diferencia. En la gráfica situada arriba observamos tres procesadores, la CPU, una tarjeta gráfica GeForce y una tarjeta gráfica Tesla, a lo largo de los años. Mientras que el ancho de banda teórico de una CPU en 2013 alcanzaba los 60 GB/s, una tarjeta gráfica Tesla K40 era capaz de llegar a los 285 GB/s, y si nos fuéramos a la GeForce 750 Ti, más de 300 GB/s. [Datos obtenidos del año 2013]

El conjunto de software de CUDA se compone de varias capas, empezando por un controlador de dispositivo, una API y las bibliotecas primarias, que van incluidas en la descarga del soporte para CUDA. Una de las bibliotecas más importantes y de la que hablaremos más tarde es BLAS, una biblioteca dedicada al cálculo matemático. (NVIDIA España 2015) (NVIDIA Developer 2015)

7.1. Diferencias entre una CPU y una GPU

Existen una gran cantidad de diferencias entre una CPU y una GPU, entre ellas, el ancho de banda y el número de operaciones por segundo, visto en el apartado anterior. En este apartado nombraremos unas cuantas diferencias más entre estos dispositivos, más en profundidad.

Debido a este tipo de diferencias la GPU es una mejor opción para el cómputo de operaciones en paralelo que la CPU, y por lo tanto, para la realización de operaciones.

Una CPU dispone normalmente de entre 4 y 8 núcleos o procesadores. Estos procesadores suelen ser de alto rendimiento, de entre 2 y 12 GHz cada uno. Cada uno de estos núcleos físicos es visto por el sistema operativo como dos núcleos lógicos, es decir, que en realidad es como su tuviéramos entre 8 y 16 núcleos o hilos. A esta propiedad se le denomina *Hyper-Threading*.

Una GPU dispone de múltiples núcleos (256, 512…). Estos procesadores tienen un rendimiento inferior a los de una CPU. Normalmente actúan en torno 1 GHz, sin embargo, existe un mayor número dentro de cada tarjeta.

En la CPU se disponen tres niveles de caché, desde la más interna y así mismo rápida, L1, que suelen ser unos 32kB de datos y otros 32 kB de instrucciones, la caché L2, de 256 kB, y una caché L3 compartida entre todos los núcleos de hasta 15 MB.

En una GPU se disponen únicamente de dos niveles de caché, una primera caché L1 más interna y rápida de entre 16 y 96 kB (desde las de primera generación hasta las actuales) que se reparten entre datos e instrucciones y una caché L2 externa de 1492 kB por ejemplo para una Kepler GK110, o de 2048 kB para una Maxwell GK204. [Datos obtenidos del año 2013]

La peticiones a memoria hechas en la CPU se gestionan de forma independiente por cada hilo, esto quiere decir que si hubiera X hilos pidiendo un mismo dato de una posición de memoria, se tardaría X veces el tiempo de un hilo, y por tanto X veces el tiempo de petición a memoria.

Las peticiones a memoria hechas en una GPU se hacen por grupos de hilos. Esto hace que el acceso a memoria sea mucho más rápido si existen varios hilos que buscan un mismo dato. Los accesos se suelen hacer en grupos de 32 hilos. Después, otros 32 hilos pueden buscar otro dato distinto. La gran ventaja es el poder realizar una misma operación en cadena con distintos datos.

Si tuviéramos que referenciarnos a la Taxonomía de Flynn, lo más semejante al cálculo GPGPU sería el SIMD, es decir, *Single Intruction Multiple Data*, o lo que es lo mismo una misma instrucción para un mismo dato. Pero siendo un poco más precisos, el modelo exacto sería el SIMT, es decir, *Single Instruction Multiple Thread*, con lo que queremos decir que es una misma instrucción para varios hilos. Se podría decir que es un híbrido entre SIMD y MIMD Simétrico (*Multiple Instruction Multiple Data*).

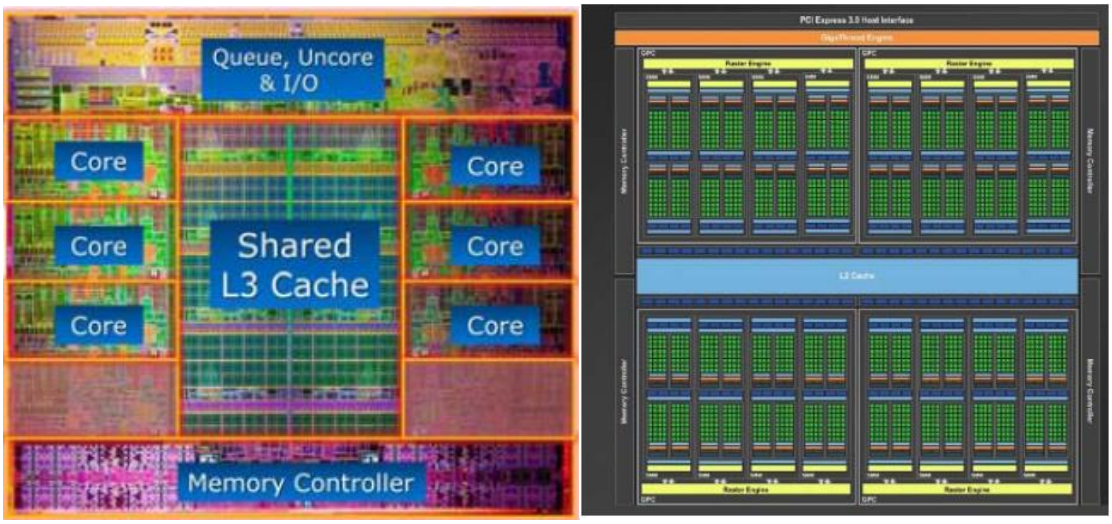


Figura 7.2. Comparación Intel Core I7-3960x con Maxwell GK204 (NVIDIA Developer 2015)

Como podemos observar en la imagen, el procesador Intel Core I7 tiene 3 memorias caché, y un total de 6 núcleos (aunque puede ampliarse a 8). Mientras que en la tarjeta gráfica Maxwell GK204 podemos observar que solo hay dos cachés en color azul, y hay múltiples núcleos, en color verde.

En esta última imagen hemos podido observar como la GPU da mayor importancia a los núcleos de computación, mientras que la CPU da una mayor importancia a la caché. Sin embargo, la pregunta es, ¿Qué porcentaje de la oblea de silicio ocupa cada parte en cada uno de los dispositivos? Para ello, observamos la siguiente imagen:

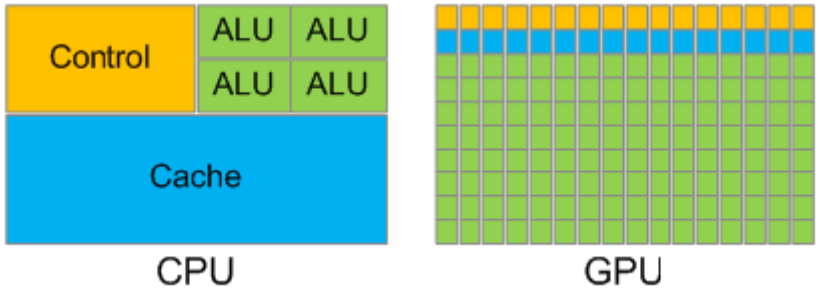


Figura 7.3. Comparación espacio físico CPU con GPU (Martínez Zarzuela 2015)

Podemos observar como en la CPU en una menor parte de la carga es dedicada al control, al igual que a la unidad aritmético – lógica, o lo que es lo mismo, a la parte de cálculo, y finalmente la otra mitad está dedicada a una amplia jerarquía de memoria caché. Esto resulta útil si tenemos datos a los que queremos acceder rápido, ya que vamos a acceder muchas veces a ellos.

Sin embargo, en la GPU sólo una pequeña parte está dedicada al control, otra pequeña parte está dedicada a la memoria caché y el mayor porcentaje de la oblea de silicio son ALUs dedicadas al cálculo computacional. Esto hace que la capacidad computacional de una GPU sea mucho mayor que en una CPU. Esto resulta útil si tenemos que realizar operaciones muy costosas con poca cantidad de datos, de forma que no sea necesario acceder a memorias externas.

Observando esta imagen podríamos extrapolar que, si necesitamos buscar datos en memoria, deberíamos usar la CPU. Si posteriormente queremos realizar operaciones con esos datos, deberíamos usar la GPU. Para ello será necesaria una forma de traspasar los datos de la memoria de la GPU a la memoria de la CPU. (NVIDIA Developer 2015) (Martínez Zarzuela 2015)

7.2. Paralelismo de datos

Se ha hablado durante el trabajo del paralelismo de datos, sin embargo, no se ha dado una explicación sobre en qué consiste este término. En esta sección tendrá lugar una explicación detallada.

El paralelismo de datos se refiere principalmente a la propiedad de un programa donde varias operaciones aritméticas pueden ser realizadas de una forma segura y sin errores de forma simultánea. Un ejemplo sencillo para ilustrar este fenómeno puede ser la multiplicación de matrices.

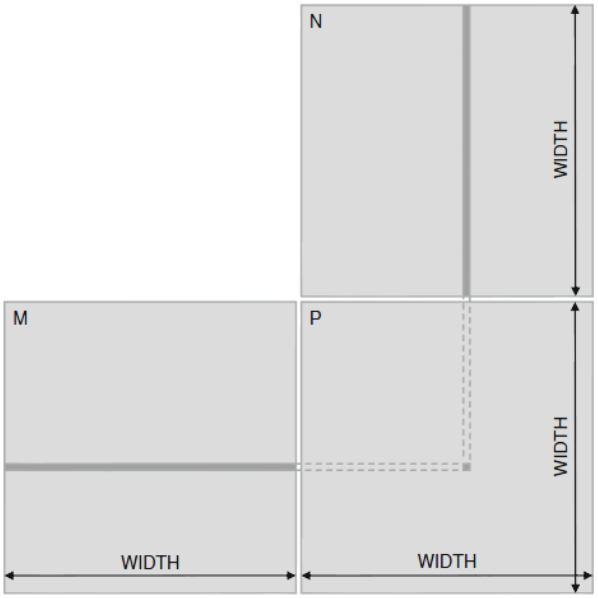


Figura 7.4. Ilustración multiplicación de matrices (Kirk, Hwu 2010)

Podemos observar como en la multiplicación de matrices, para obtener un punto de la matriz P, necesitamos una fila entera de la matriz M y una columna entera de la matriz N. (Estamos multiplicando M x N). Supongamos que el punto marcado en la imagen es el P_{ij}, con i el subíndice de fila y j el subíndice de columna. Este punto se obtendría calculando:

$$Pij = Mi1 * N1j + Mi2 * N2j + Mi3 * N3j + Mi4 * N4j + Mi5 * N5j \ldots \quad (7.1)$$

Cada una de las operaciones $M_{iX} * N_{Xj}$ puede ser realizada de forma independiente al resto, por lo tanto se podrían hacer todas a la vez, es decir, en paralelo. Después, se suman todos los términos y obtendríamos nuestro resultado. Es importante que se espere a que todas las operaciones hayan terminado, ya que si no, el espacio de memoria donde se supone que debería estar guardado el resultado puede ser erróneo. Por ello, es importante ocultar la latencia cuando

hablamos de paralelismo. Esto es a lo que se refiere uno cuando al definir el paralelismo de datos se incluyen las palabras “de forma segura”.

Otro ejemplo de paralelismo lo podemos encontrar también en la misma multiplicación de matrices. Podemos observar que cada punto P_{ij} es independiente de los demás. Por lo tanto, no es necesario hacerlo de forma secuencial (ya que las matrices M y N no cambian) como lo haría una CPU. Se pueden hacer todos los cálculos en paralelo. Esto puede ser un poco ineficiente para matrices pequeñas, sin embargo, para matrices grandes, se ahorra muchísimo tiempo. Por eso ahora el uso de las tarjetas gráficas de propósito general (GPGPU) está en auge. (Kirk, Hwu 2010)

7.3. Lenguaje CUDA

CUDA pretende explotar el paralelismo de las tarjetas gráficas, por eso existen gran cantidad de instrucciones dedicadas a ello. Además, es necesaria una comunicación entre la CPU y la GPU. Esto es algo que también el lenguaje CUDA abarca. Comenzaremos con una pequeña introducción a este lenguaje.

Como hemos explicado en la conclusión del apartado anterior, dispondremos de dos dispositivos, una CPU, donde correrá al inicio nuestro programa, y una GPU, que actuará como coprocesador y estará controlada por la CPU. La CPU se encargará de accesos a memoria, búsqueda de datos, y todo lo que tenga que ver con la memoria caché, mientras que toda la carga computacional la realizará la tarjeta gráfica. Para ello será necesario que los datos de la memoria de la CPU pasen a la memoria de la GPU a través de unas sentencias de las que hablaremos posteriormente.

La CPU se denominará *host*, y la GPU la llamaremos *device*. Esta GPU será pasiva, es decir, no podrá llamarse a sí misma, únicamente podrá ser llamada a través de la CPU. Actualmente, en las últimas actualizaciones del lenguaje CUDA, exactamente a partir de CUDA 5.0, se ha conseguido acceder al paralelismo dinámico, de forma que un device sí que puede llamar a otro device para actuar en paralelo. Sin embargo, a lo largo de este proyecto no se va a contemplar esa posibilidad.

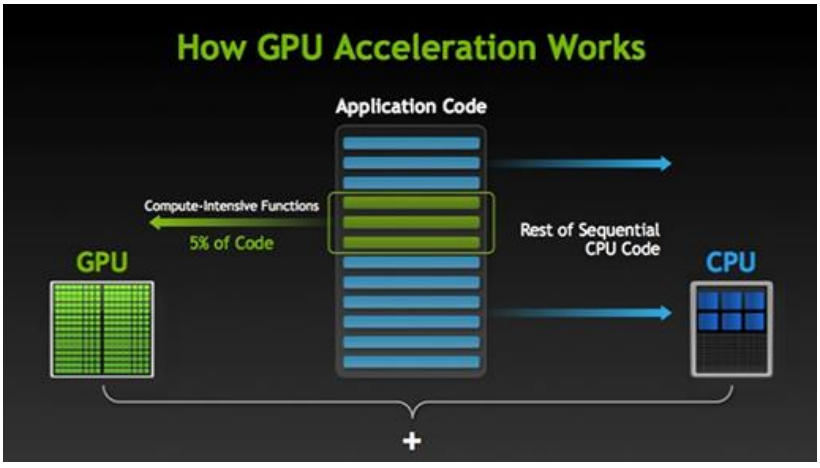


Figura 7.5. División del código entre CPU y GPU (NVIDIA Developer 2015)

Cuando ejecutemos un programa, habrá líneas de código que serán ejecutadas en el host, y otras líneas de código que serán ejecutadas en el device. Las líneas de código ejecutadas en el device serán llamadas por el host, a través de módulos o *kernels* de CUDA. Desde estos kernels

tendremos acceso a las memorias de la GPU. En las sentencias de la GPU se realizarán los cálculos matemáticos deseados por el programa. El código del host se centrará en la gestión de memoria, tanto de reserva como de liberación, en la invocación de los kernels, en el intercambio de datos entre las memorias CPU y GPU y en el posterior procesado de los datos obtenidos.

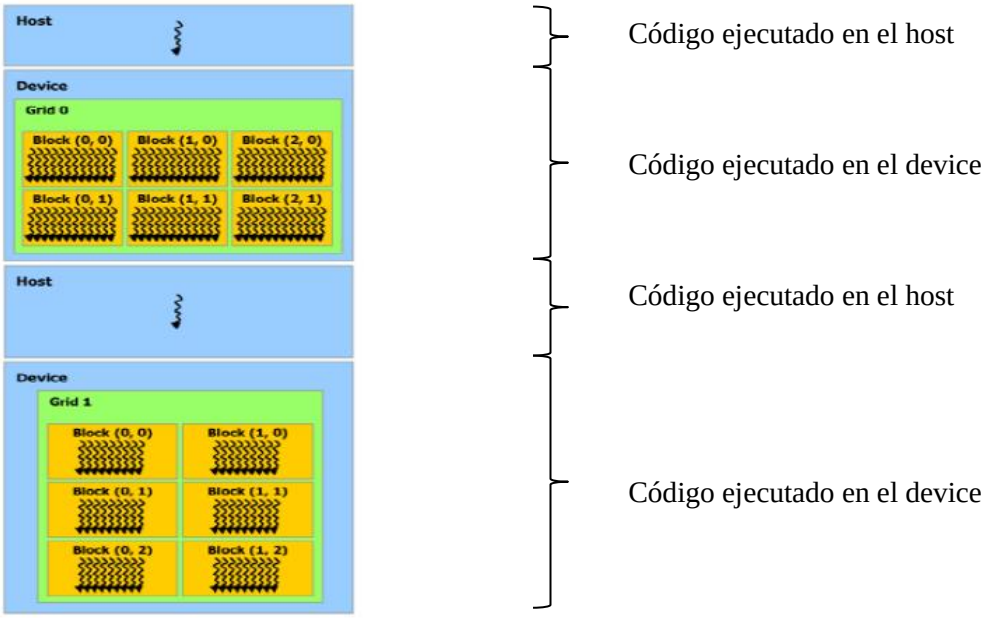


Figura 7.6. Ejemplo de desarrollo host – device (Elaboración propia a partir de Martínez Zarzuela 2015)

En la figura anterior podemos observar el desarrollo normal de un programa donde se quiere realizar un cálculo con una GPU. Primero se lanza el programa en el *host*, donde sólo hay un hilo. A continuación el host llama a un kernel que hace que el programa quede a la espera del device, que ejecutará su parte de código. El kernel lanzará un *grid* que contendrá unos bloques. Algo que veremos posteriormente. Finalmente, cuando el *device* termina, de nuevo pasa el control al host, hasta que este lanza un nuevo kernel para volver a realizar operaciones en el device. Así continuamente, hasta que el programa finaliza. El programa finalizará siempre con el control en el host.

Anteriormente hemos estado hablando sobre *grids*, bloques e hilos. Pero, ¿a qué se refieren exactamente estos términos?

Un hilo o *thread* en inglés es la unidad básica de operación. Dependiendo del número de hilos que pueda haber simultáneamente en ejecución una tarjeta tendrá mayor o menor capacidad de paralelismo. Estos hilos se agrupan en bloques o *blocks*. Estos bloques representan diferentes niveles de paralelismo a nivel lógico. Finalmente, varios bloques del mismo tamaño se agrupan todos en una única malla o *grid*. Este grid puede ser ejecutado por el kernel, el programa que es lanzado desde el host para ser ejecutado en el device. El tamaño de nuestro grid será imprescindible para ajustar la eficiencia del programa en función del paralelismo que podemos crear. Un *warp* es otra forma de agrupación de los hilos, en esta forma los hilos se unen en grupos de 32 unidades. Un bloque debería tener un tamaño siempre de un múltiplo de 32 hilos, o lo que es lo mismo, de un número entero de warps.

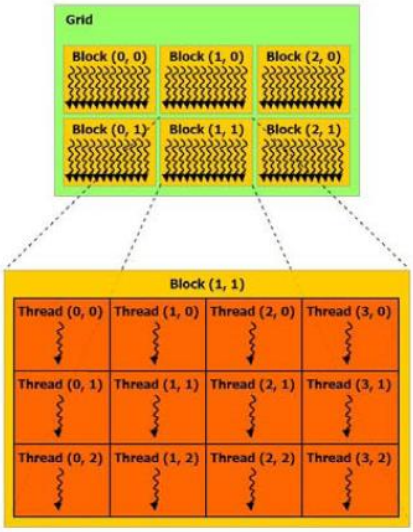


Figura 7.7. Distribución de un grid en bloques e hilos (Martínez Zarzuela 2015)

En la figura 7.7 podemos observar un grid que ha sido lanzado por un kernel. Nuestro grid tiene una dimensión de 6 bloques, dos en el eje Y y tres en el eje X. Cada uno de estos bloques se divide en 12 hilos, tres en el eje Y y cuatro en el eje X. Un grid como este sería idóneo para operaciones con matrices con una dimensión de 6 filas (3 del primer bloque y otros 3 del segundo) por 12 columnas (3 bloques por 4 hilos cada bloque). Por ello, dependiendo de las matrices con las que estemos trabajando el grid deberá cambiar de tamaño, para así obtener la mayor eficiencia en cuanto al paralelismo se refiere.

Tras observar la distribución de nuestro grid, nuestra siguiente pregunta es: ¿Cómo hacemos referencia al grid que hemos declarado? Para ello existen en CUDA ciertas variables ya integradas de las que hablaremos a continuación.

- `typedefstruct {…} dim3;` Es un tipo predefinido en CUDA de tres dimensiones (x,y,z).
- `dim3 threadIdx`: Es una variable integrada con el identificador del hilo.
- `dim3 blockIdx`: Es una variable integrada con el identificador del bloque.
- `dim3 blockDim`: Es una variable integrada con la dimensión el bloque.
- `dim3 gridDim`: Es una variable integrada con la dimensión del grid.

Observando la figura 5, podríamos decía que la variable `gridDim` vale (3,2), la variable `blockDim` vale (4,3) y dependiendo del bloque o el hilo en el que estemos las variables `blockIdx` y `threadIdx` valdrán una cosa u otra, siempre indexándose como un vector de tres unidades (x,y,z). En el caso de que estemos ante un grid bidimensional, una de estas unidades (normalmente z) valdrá 1.

```
unsigned int tidx =
     threadIdx.x + blockDim.x*blockIdx.x;
```

Figura 7.8. Cálculo del índice absoluto de un hilo para un grid de hilos lineal. (Elaboración propia)

Estas son las variables que el host deberá indicar al device a la hora de lanzar un kernel. La rutina básica para lanzar un kernel es:

```
Kernel<<<gridDim,blockDim>>>(args,…);
```

O en el caso de utilizar plantillas (*templates*) sería:

```
Kernel<template><<<gridDim,blockDim>>>(args,…);
```

Este kernel no va a devolver nada, siempre será de tipo *void*. Se declara con el calificador `__global_`, y normalmente siempre utiliza la memoria de la GPU, aunque también puede utilizarse una parte de la memoria de la CPU si ésta es mapeada. (Martínez Zarzuela 2015) (NVIDIA Developer 2015) (Sanders, Kandrot 2011) (CUDA Toolkit 2015)

7.4. Un primer programa

Un ejemplo de un primer programa sencillo puede ser el típico "Hello World" que se hace en todos los lenguajes de programación. Como ya hemos dicho, está basado en C++, por lo tanto tomaremos eso como punto de partida:

```
int main(void){

    printf("Hello World!\n");
    return 0;
}
```

Figura 7.9. Un primer programa (Elaboración propia)

A continuación incluimos la llamada al kernel y la declaración de éste. Declaramos el kernel como `__global__`. El tamaño del bloque el tamaño del grid serán de 1 unidad, ya que no vamos a realizar ninguna operación, por lo tanto únicamente habrá que añadir la línea de código `kernel<<<1,1>>>();`

```
#include <iostream>
#include <stdio.h>
#include "cuda_runtime.h"

__global__ void kernel(void){

}

int main(void){
    printf("Hello World!\n");
    kernel<<<1,1>>>();
    return 0;
}
```

Figura 7.10. Un primer programa añadiendo un kernel (Elaboración propia)

El siguiente paso al que debemos llegar es el de pasar parámetros desde nuestro host a nuestro device y viceversa. Esta tarea será realizada por el host.

Para ello, será necesaria la reserva de memoria, tanto en el host como en el device, ya que si no obtendremos un error de segmentación, o lo que es lo mismo, un error de acceso a memoria que no ha sido reservada.

Para la reserva de memoria utilizaremos dos funciones. En el host, usaremos `malloc()`. La función dada por C++, mientras que para el device, utilizaremos `cudaMalloc()`. Una función similar pero para reservar memoria en el device.

Posteriormente, aparte de reservar la memoria, habrá que liberarla. De nuevo, con la función de C++ `free()` se liberará en el host, y análogamente en el device con `cudaFree()`.

Un dato muy importante a destacar es que estas funciones actúan con punteros a la memoria. Es decir, uno pide que se reserven una cantidad de bytes a partir del siguiente punto de la memoria, y el programa se lo concede. Por ello, las variables con las que trabajaremos serán punteros a direcciones de memoria, o incluso punteros a punteros a direcciones de memoria.

El paso de parámetros se hará como en una función normal. Tras llamar al kernel, dentro de los paréntesis se añadirán en orden los parámetros a enviar, mientras que en la declaración del kernel habrá que añadir el tipo de dato que se debe recibir.

El traspaso de datos de la memoria del host a la memoria del device y viceversa se hará con la función `cudaMemcpy()`. Esta función copiará los datos de un lado a otro, y en el último argumento de esta función se indicará si se desea copiar del host al device, del device al host, del host al host o del device al device.

A continuación se ilustrará todo lo enunciado con el ejemplo de un código sencillo. Será la suma de dos números, por lo tanto tampoco se aprovechará el paralelismo. Nuestro tamaño de grid y de bloque será de nuevo 1.

Incluiremos en este código una macro muy habitual en el lenguaje CUDA. Es una macro de detección de errores. Normalmente, se devuelve un código de error que a través de la función `cudaGetErrorString(err)` se puede saber con palabras cual es el problema en cuestión. Sabiendo que esta macro se puede utilizar en todas las funciones CUDA, nos será muy sencillo obtener el error en caso de que lo haya, y saber exactamente dónde y por qué ha ocurrido.

La macro se llamará normalmente `CUDA_CALL()` o `HANDLE_ERROR()`.

A continuación se muestra el código descrito

```
#include <iostream>
#include <stdio.h>
#include "cuda_runtime.h"


#define CUDA_CALL(x) do{\
    cudaError_t err = (x);\
    if(err!=cudaSuccess){\
        printf("Error \"%s\"\n",cudaGetErrorString(err));\
        exit(-1);\
    }}while(0)

__global__ void add(int a,int b, int *c){
    *c = a+b;
}

int main(void){
    int c;
    int *dev_c;

    CUDA_CALL(cudaMalloc((void **)&dev_c,sizeof(int)));
    add<<<1,1>>>(2,7,dev_c);

    CUDA_CALL(cudaMemcpy(&c,dev_c,sizeof(int),cudaMemcpyDeviceToHost));
    printf("2+7=%d\n",c);
    cudaFree(dev_c);
    return 0;
}
```

Figura 7.11. Suma de dos números (Elaboración propia)

Podemos observar, tras los `includes`, la macro definida. Esta macro nos devuelve un error y para nuestro programa en el caso de que lo que se encuentra dentro no devuelva `cudaSuccess`, es decir, que no devuelve un calor correcto sin errores.

A continuación observamos nuestro kernel. Ésta vez sí que hay variables dentro de él. Como se ha dicho, se va a realizar la suma de dos números, y esto es lo que el kernel hace. Toma a y b, los suma, y los introduce en c.

Dentro de nuestra función `main()` observamos la declaración de dos variables, la variable `c` que será el destino de la suma en el host, y la variable `dev_c`, que será el destino de la suma en la memoria del device. La variable `dev_c` debe ser definida como un puntero a memoria.

A continuación se reserva la memoria en el device, los dos argumentos de la función `cudaMalloc()` son el punto de partida y el espacio requerido.

Posteriormente se realiza la llamada al kernel, ahora sí, con los valores pasados por referencia. Se ha decidido sumar los números 2 y 7 como ejemplo, y posteriormente introducirlos en el espacio de memoria reservado en el device.

Con el dato en el device, debemos traspasarlo al host a través de la función `cudaMemcpy()` para imprimirlo por pantalla. Como queremos pasarlo del device al host, el último argumento será la opción *`cudaMemcpyDeviceToHost`*.

Finalmente, tras imprimirlo por pantalla, liberamos la memoria del device. (Sanders, Kandrot 2011) (CUDA Toolkit 2015)

7.5. Explotando el paralelismo de las GPUs

Tras un primer inicio al lenguaje CUDA, estamos ya preparados para empezar a explotar el paralelismo de nuestra tarjeta gráfica. Comencemos con un ejemplo sencillo, y avanzaremos con ejemplos más complicados.

Se ha visto en el apartado anterior la suma de dos números. Procedamos ahora a la suma de vectores. El procedimiento será el mismo excepto por un par de variantes. Ahora los tamaños del bloque y del grid cambiarán en función del tamaño del vector, utilizaremos las funciones `cudaSetDevice()` y `cudaDeviceReset()` y el código del kernel será un poco más complejo.

Para elegir la dimensión del bloque y del grid, lo primero que debemos pensar es a qué nos enfrentamos. Una suma de vectores es en una dimensión, eso quiere decir que tanto el grid como el bloque también lo serán. El tamaño del bloque lo seleccionamos nosotros. Debe ser múltiplo de 32, como se dijo anteriormente, para que incluya un número entero de warps. Sin embargo, el tamaño del grid viene en función del tamaño del vector y del tamaño de bloque escogido. Se deberá seleccionar un grid adecuado, por ello se suele utilizar el tamaño del vector dividido entre la dimensión del bloque. De esta forma, se asegura que un grid completo tenga al menos la dimensión del vector a sumar.

```
int blockdim = 256;
int griddim = (N-1)/blockdim+1;
```

Figura 7.12. Cálculo de las dimensiones de bloque y grid (Elaboración propia)

Las funciones `cudaSetDevice()` y `cudaDeviceReset()` sirven para seleccionar un device y para resetearlo tras las operaciones. En los ordenadores donde haya más de una tarjeta gráfica, cada CPU estará determinada por un número. Si deseamos trabajar con la primera, debemos seleccionar la número 0.

En el código del kernel debemos empezar a explotar el paralelismo. Para ello utilizaremos las variables de las que hemos hablado anteriormente. En particular, utilizaremos `threadIdx` y `blockIdx`. Los identificadores de hilo y bloque.

Cuando se llame al kernel, tendremos múltiples bloques con múltiples hilos que tendrán que hacer lo mismo sobre diferentes datos. Nuestro objetivo es que cada hilo sume dos números, de forma que con una sola pasada todos los hilos hayan sumado todos los números de los vectores y todo se haga rápidamente. Para ello, a cada hilo hay que indicarle que sume un número distinto. Por lo tanto, tendrá que haber una variable que representa la posición en el vector, y que sea diferente para cada hilo. A esa variable la hemos llamado `tidx`. ¿Cómo sabemos que tidx tiene cada hilo? Basta con calcular la posición total dentro del grid de cada hilo. Si tenemos bloques de X tamaño, multiplicamos nuestro identificador de bloque por X, y le sumamos el identificador de hilo. A continuación, únicamente tenemos que sumar los dos números que estén en esas posiciones de memoria.

Podemos observar en el código del *Apéndice 1.1 "Suma de vectores"* cómo se realizan los pasos anteriores del mismo modo, aunque ampliando a un vector en lugar de un número solo.

La declaración de variables es similar, ahora declarando `A`, `B` y `C` como vectores. Es importante destacar que `dev_A`, `dev_B` y `dev_C` siguen siendo punteros, esta vez el inicio del vector en la memoria del device. Copiamos los datos a la memoria del device y llamamos de nuevo al kernel para que nos haga la suma en la GPU. Finalmente copiamos el resultado de la memoria de la GPU a la memoria de la CPU y lo imprimimos por pantalla.

También puede resultar muy útil indicar que en el caso de haya hilos que no tengan que hacer nada (porque el vector es inferior al tamaño del grid ya que el grid debería ser siempre múltiplo de 32) no hagan nada. Esto se puede hacer con un simple `while`. Finalmente, quedaría el código del kernel de esta forma:

```
__global__ void vectorAdd(int *A,int *B, int *C)
{
    //int tidx = blockIdx.x;
    int tidx = threadIdx.x + blockIdx.x*blockDim.x;
    while (tidx < N){
    C[tidx] = A[tidx] + B[tidx];
    }
}
```

Figura 7.13. Kernel limitando el número de elementos al tamaño del vector (Elaboración propia)

Pero no vamos a quedarnos únicamente en la suma de vectores. Pasemos a un ejemplo con el que poder explotar de forma más clara el paralelismo de las GPUs, y poder observar una eficiencia mayor en las simulaciones. Pasemos a la multiplicación de matrices.

La multiplicación de matrices ya no es tan sencilla como la suma de vectores. En este caso hay que considerar más factores que una simple suma, ya que tenemos términos en el eje X y términos en el eje Y que debemos ir multiplicando entre sí para luego sumarlos.

Aunque no es obligatorio, utilizaremos plantillas o ***templates***. Un template es una característica de C++ que permite que haya variables genéricas, es decir, que no sea necesario ir cambiando el tipo de variable constantemente, sino que es el propio C++ el que identifica el tipo de variable que debe ser. Esto no será muy útil con el tamaño de bloque, ya que se usará mucho en la multiplicación de matrices.

En la multiplicación de matrices también utilizaremos memoria compartida. Aunque es algo de lo que hablaremos más adelante, se hará una breve introducción. Utilizando memoria compartida para almacenar las matrices en el device hará que nuestro programa sea considerablemente más rápido. Ocultará la latencia del orden de 100 veces mejor que con la memoria normal. Además, será necesaria una sincronización de todos los hilos, para que no haya problemas en la búsqueda de los datos.

Por último, se hará un análisis del rendimiento de CUDA respecto a C++, que podremos ver en las posteriores secciones del trabajo. En particular, se analizará el tiempo que tarda en realizar las operaciones y el número de operaciones en punto flotante por segundo (FLOPS). Por ello, se utilizan distintas funciones de creación de eventos. Estas funciones son:

- `cudaEventCreate(X);` Crea un evento de tipo `cudaEvent_t`.
- `cudaEventRecord(X,0);` Almacena el tiempo en el evento `X` de tipo `cudaEvent_t`.
- `cudaEventSynchronize(X);` Sincroniza un evento `X` de tipo `cudaEvent_t`.
- `cudaEventElapsedTime(time,X,Y);` Almacena en la variable "`time`" la diferencia temporal entre los evento `X` e `Y` de tipo `cudaEvent_t`.

Se puede observar en el *Apéndice 1.2 "Multiplicación de matrices en CUDA"* el código de la multiplicación de matrices desarrollado sobre CUDA y C++.

Inicialmente incluimos los *runtime* de CUDA y el resto de librerías para poder trabajar con todas las funciones, también definimos la macro de la que hablamos anteriormente.

La función `matrixMulCUDA()` se encarga de hacer la multiplicación de matrices en el device. El objetivo es que, a través de un bucle `for`, se vayan guardando término a término los datos deseados en la memoria compartida, ya que como hemos dicho, es más rápida. Posteriormente sincronizamos todos los hilos para asegurar que todos tienen la multiplicación que deseamos y que no haya algún valor que por cualquier motivo aún no se haya multiplicado. Finalmente incluimos en nuestra matriz C la multiplicación hecha desde la memoria compartida, y volvemos a sincronizar los hilos.

La siguiente función es la función de multiplicación de matrices hecha en la CPU. Esta función se ha hecho para poder comparar tiempos entre la GPU y la CPU, pero no se va a hablar de ella más que lo expresamente necesario.

Se realiza también una función de comparación entre dos vectores o matrices, a modo de poder comprobar que la multiplicación de matrices se ha hecho correctamente.

En cuanto al código de la función main, es importante destacar que hay variables dedicadas únicamente a la simulación. Un ejemplo de esto son las variables `nIter` y `average`. La variable `nIter` nos indica el número de iteraciones que se desean hacer, cada iteración tendrá un valor de las matrices distinto. `Average` nos indica el número de veces que se va a realizar la simulación para cada matriz, para luego calcular el tiempo medio.

Declaramos las variables tanto para el host como para el device, elegimos los tamaños de nuestras matrices, reservamos la memoria y las inicializamos a los valores deseados. Elegimos valores aleatorios cercanos a cero para que la multiplicación sea más sencilla y no de errores ya que si los números son grandes es posible saturar la capacidad del ordenador. El rango para un dato de tipo *float* va desde [-3.4 exp(38), -1.4 exp(-45)] en los negativos hasta [1.4 exp(-45), 3.4 exp(38)] en los positivos. Esto son números en principio muy grandes, sin embargo, si tenemos que multiplicar matrices de 1000 x 1000 al final acaban resultando pequeños.

Declaramos y creamos los eventos de recogida de tiempos tanto de la CPU como de la GPU, y comenzamos a multiplicar en la GPU. Tras obtener la dimensión de los bloques y del grid, esta vez siendo variables dim3 ya que ahora no es sólo una dimensión, procedemos a lanzar el kernel para multiplicar las matrices que hemos definido. Finalmente recogemos el tiempo tras la multiplicación y calculamos la diferencia de tiempos. Realizamos el mismo procedimiento para la CPU.

En la última parte se puede observar los cálculos de tiempo, y FLOPS de cada simulación, para después calcular la eficiencia de multiplicar en GPU en lugar de en CPU.

Finalmente destruimos los eventos y liberamos la memoria. (Sanders, Kandrot 2011) (CPLUSPLUS 2015) (CUDA Toolkit 2015)

8. Librería cuBLAS

La librería cuBLAS es una implementación de subrutinas básicas de álgebra lineal (Basic Linear Algebra Subroutines) desarrollados en el entorno para CUDA. Permite al usuario una mayor explotación de los recursos de las GPUs. El funcionamiento de éste es similar al de CUDA, explota el paralelismo de los datos. Sin embargo, las funciones incluidas en cuBLAS, además de ser más específicas, son más eficientes ya que están dedicadas expresamente al cálculo de álgebra lineal.

Soporta distintos tipos de datos: simple, double, float y complex, además del uso para múltiples GPUs.

Esta librería es capaz de trabajar hasta 17 veces más rápido que la MKL BLAS, otra librería básica (Math Kernel Library BLAS) de CPU. Con lo que se puede decir que la libraría cuBLAS es bastante más eficiente.

Observando la figura siguiente podemos ver la diferencia de GFLOPS.

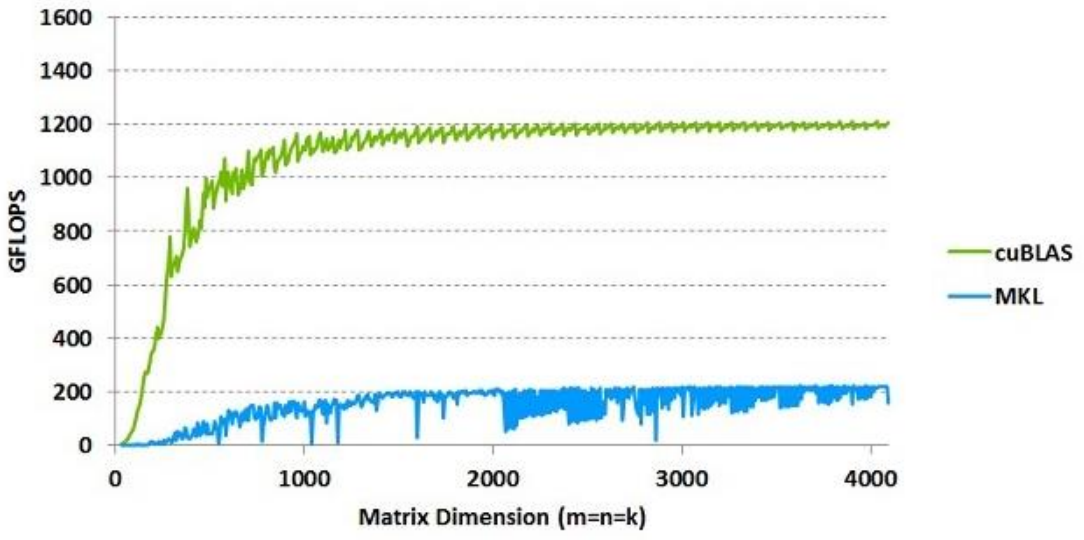


Figura 8.1. Relación cuBLAS con MKL BLAS (NVIDIA Developer 2015)

Podemos observar como la librería cuBLAS es capaz de alcanzar los 1.200 GFLOPS, mientras que la librería MKL BLAS sólo llega a los 200. Es una gran diferencia en términos de eficiencia. (NVIDIA Developer 2015) (NVIDIA 2015)

8.1. Multiplicación de matrices con cuBLAS

Con el objetivo de dar uso a esta librería se va a realizar la multiplicación de matrices. A priori debería ser más sencillo de programar, ya que tenemos una librería, y debería ser más eficiente.

Para poder utilizar cuBLAS, además de tener que incluir los archivos de cabecera de esta librería, es necesario que en las opciones de construcción, en el enlazador NVCC haya que incluir la librería cuBLAS también.

Pudiendo usar los recursos de cuBLAS, tenemos a nuestro alcance ciertas funciones espejo a las de CUDA. Por ejemplo, `cublasSetMatrix()`, `cublasSetVector()`, `cublasGetMatrix()` o `cublasGetVector()`.

Estas funciones sirven para copiar tanto vectores como matrices desde el host hasta el device y viceversa, como se hacía con `cudaMemcpy()`.

Además, se crearán variables de estado de tipo `cublasStatus_t` y `cublasHandle_t`.

Pero el verdadero quid de la cuestión se centra en la función `cublasSgemm()`. Esta función sirve para multiplicar matrices. Mientras que antes con CUDA teníamos que hacer nuestro propio kernel y lanzarlo, ahora únicamente tenemos que invocar a una función que se encuentra dentro de la librería cuBLAS e introducir los parámetros en orden.

Para ver el código de la multiplicación de matrices en cuBLAS, diríjase al *Apéndice 1.3. "Multiplicación de matrices en cuBLAS".*

Inicialmente, aparte de las variables `Niter` y `average` declaradas para la simulación al igual que en el ejemplo de CUDA, declaramos el tamaño de nuestra matriz. Esta vez, creamos un

constructor denominado `matrix_size` en el que introducimos los valores de las matrices A, B y C.

Posteriormente reservamos la memoria para el host de la misma forma que en CUDA. Sin embargo, para la reserva en el device utilizaremos comandos distintos. Antes de iniciar la reserva en el device iniciamos nuestro contador de evento, ya que toda esta reserva ya se incluye dentro de las rutinas que afectan a la GPU y por lo tanto deben contar dentro del tiempo calculado para poder posteriormente obtener la eficiencia del programa.

A continuación se reserva la memoria en el device y se copian los datos del host al device con `cublasSetMatrix()`. Esta rutina, específica de cuBLAS, es más rápida que el `cudaMemcpy()`,ya que la reserva de memoria y la colocación de ésta se hace de forma más eficiente. Por lo tanto será otra ventaja a la hora del cálculo de tiempo. De todas formas, se pueden usar ambas rutinas para la traslación de los datos entre dispositivos.

A continuación se declaran dos variables auxiliares, alfa y beta, que sirven para operar con la función `cublasSgemm()`.

`cublasSgemm()` es una función de nivel 3 de la biblioteca de cuBLAS. Una función de nivel 3 implica que se realizan operaciones de matrices con otras matrices. Mientras que las de nivel 2 son de vectores con matrices y las de nivel 1 de vectores o de matrices únicamente.

Esta función realiza la multiplicación de matrices como tal, es decir, realiza el código que hemos realizado anteriormente en CUDA. Sin embargo, la capacidad de almacenamiento y de búsqueda de los datos en esta rutina es claramente superior a la capacidad del programa que habíamos realizado nosotros. Por ello, las ventajas de utilizar este código serán considerables.

Esta función realiza la siguiente operación:

$$C = \alpha * A * B + \beta * C \tag{8.1}$$

Donde alfa y beta son constantes, y A, B y C son matrices. Como solamente se está buscando C = A*B, alfa deberá valer 1 y beta deberá valer 0. El resto de valores que se le deben introducir a esta función son direcciones de memoria donde se encuentras las matrices, tamaño de filas de la matriz, y si se quiere que esta matriz se trasponga o no. Esto se hace con la opción `CUBLAS_OP_N`, que puede tomar diferentes valores. En función de lo que valga, la matriz se introducirá de una forma u otra:

$$\text{op}(A) = \begin{cases} A & \text{if transa == CUBLAS\_OP\_N} \\ A^T & \text{if transa == CUBLAS\_OP\_T} \\ A^H & \text{if transa == CUBLAS\_OP\_C} \end{cases}$$

Figura 8.2. Colocación de la matriz en función de la opción CUBLAS_OP (cuBLAS Toolkit 2015)

Finalmente, se copia la solución del device al host, se paran los eventos para calcular el tiempo, se obtiene el cálculo en CPU como en el apartado anterior y se realiza un estudio del tiempo y del número de operaciones en punto flotante por segundo. También se comprueba si el resultado

es el esperado.Se destruyen los eventos y se libera la memoria para terminar con el programa. (cuBLAS Toolkit 2015) (Chrzęszczyk, Chrzęszczyk 2015)

8.2. Otros programas con cuBLAS

Aunque la multiplicación de matrices era el objetivo principal, se han buscado otros ejemplos para poder explotar el rendimiento de las tarjetas gráficas, que tengan como base la multiplicación de matrices.

Estos ejemplos son, la resolución de sistemas lineales, la factorización LU y la inversión de matrices.

8.2.1. Resolución de sistemas lineales

Procederemos a resolver el siguiente sistema:

$$A * x = b \tag{8.2}$$

Donde A es una matriz triangular, x nuestra incógnita y b un vector.

En este programa veremos tres nuevos comandos que nos van a resultar útiles. Los dos primeros son `cublasSetVector()` y `cublasGetVector()`,utilizados para mover un vector desde el *host* al *device*, y viceversa. Funciona igual que el comando `cublasSetMatrix()` y `cublasGetMatrix()`, con la diferencia de que ahora sólo nos permite introducir un vector.

El comando realmente importante de este programa es `cublasStrsv()`.

Es un comando similar al visto anteriormente, con la diferencia de que ahora se multiplica una matriz por un vector. La matriz A puede sufrir cambios antes de realizarse la multiplicación, en función de los que la variable *transa* valga, al igual que en el apartado anterior (Figura 8.2).

Otro dato que incluimos en el comando es si la matriz A que vamos a utilizar es triangular superior o inferior, para ello, hay una opción denominada *uplo* que nos permite introducir como está organizada la matriz. Para ello, habrá que introducir uno de los dos comandos, o `CUBLAS_FILL_MODE_LOWER` o `CUBLAS_FILL_MODE_UPPER` en función de lo deseado.

Si accedemos al apédice 1.4 "*Resolución de sistemas lineales*" se podrá ver el código completo. (cuBLAS Toolkit 2015) (Chrzęszczyk, Chrzęszczyk 2015)

8.2.2. Factorización LU

Es nuevo método de resolución de sistemas lineales tiene lugar debido a que hay otro tipo de formas de resolución, como la regla de Cramer, que no resultan útiles a la hora de resolver sistemas de muchas ecuaciones. Por ello, la implementación de la factorización LU puede resultar muy útil.

Si accedemos al apédice 1.5 "*Factorización LU*" se podrá ver el código completo.

Podemos observar en el código que los métodos utilizados son similares a la resolución de sistemas lineales simple, con una variación. Ahora tenemos dos matrices, una matriz L, triangular superior, y una matriz U, triangular inferior. La multiplicación de ambas matrices daría lugar a una matriz A, como la anterior. Sin embargo, esta vez no es necesario que nuestra supuesta matriz A tenga que ser triangular. Bastará con que sea definida positiva.

Se inicializarán los vectores y matrices de la forma habitual, y se pasarán a la memoria del device. A continuación, utilizaremos la función definida anteriormente, `cublasStrsv()`, dos veces, una primera para multiplicar la matriz triangular inferior U * x, y recibiremos la salida en x, para posteriormente realizar la multiplicación L * x, siendo x el nuevo valor recibido. De esta forma calcularemos L * U * x, o lo que es lo mismo, A * x si A = L * U. Finalmente traspasamos a la memoria del host la solución recibida, de nuevo en la variable x. (cuBLAS Toolkit 2015) (Chrzęszczyk, Chrzęszczyk 2015)

8.2.3. Inversión de matrices

Otra forma de resolver sistemas de ecuaciones $A * x = b$ es invirtiendo la matriz A, de forma que tengamos que resolver lo siguiente:

$$x = A^{-1} * b \tag{8.3}$$

Para ello es necesario poder invertir la matriz A. En el siguiente ejemplo veremos cómo hacer esto. Se ha incluido una nueva macro, `cublascall()`, que hace el mismo cometido que `CUDA_CALL()`, pero orientado al desarrollo en cublas.

La función que se va a utilizar en este código es una función relativamente reciente, y bastante compleja. Se llama `cublasDgetrfBatched()`. A continuación comentamos para qué sirve.

`cublasDgetrfBatched()` es una función que en un principio nos realiza la factorización LU. Sin embargo, con la configuración adecuada de sus valores se puede invertir una matriz.

La letra "D" en mayúscula indica que vamos a trabajar con datos de tipo *double*, a diferencia de en los otros casos que hemos estado trabajando con la letra "S", que es para datos simples.

Lo complejo de esta función es que en los valores que tenemos que incluir, no introducimos un puntero a la zona de memoria donde se encuentran nuestros datos, sino un *array* con varios punteros a la zona de memoria donde se encuentran nuestros diversos datos. A este *array* lo vamos a llamar Aarray[i]. La función que realiza este método es la siguiente:

$$P * Aarray[i] = L * U \tag{8.4}$$

Siendo P la matriz de permutaciones, en el caso de que la haya pivotaje, y L y U las matrices triangulares superiores e inferiores de la factorización LU.

Si accedemos al apédice 1.6 "*Inversión de matrices*" se podrá ver el código completo.

Dentro del código `main()` podemos observar la inicialización de la matriz que vamos a invertir. Podemos declarar cualquier tiepo de matriz. Desde `main()` se llama a la función `cublas_inv()`, y empezamos el código para invertir la matriz.

Declaramos e inicializamos las variables necesarias, y reservamos espacio en la memoria tanto del *host* como del *device*. Indicamos una variable denominada "*Batchsize*" o tamaño del lote, que servirá en resumen para determinar el tamaño del bloque y del *grid* con el que nuestro programa trabajará. Enviamos nuestra matriz al *device* y ejecutamos la función `cublasDgetrfBatched()` mencionada anteriormente. Finalmente, recuperamos los dos vectores de salida, uno con la matriz invertida (la recibimos en forma de vector aunque luego nosotros la trataremos como una matriz) y otro número con información sobre el éxito de la inversión o detalles como que alguna matriz es singular y no puede ejecutarse el comando.

Finalmente, devolvemos el valor de la matriz a `main()` para escribirlo por pantalla. (cuBLAS Toolkit 2015) (Chrzęszczyk, Chrzęszczyk 2015)

9. Estudio del rendimiento C++/CUDA/cuBLAS

Se han hecho diferentes pruebas con diferentes tipos de matrices para observar el rendimiento tanto en tiempo como en GFLOPS entre los distintos casos estudiados, es decir, ejecutándose el programa con C++, ejecutándose con CUDA y ejecutándose con cuBLAS.

El programa en cuestión que se ha realizado para analizar el rendimiento es el de la multiplicación de matrices. Se han creado dos códigos, uno comparando CUDA con C++ y otro comparando cuBLAS con C++. Esto se ha simulado en dos ordenadores con tarjetas gráficas distintas. En una teníamos tres NVIDIA GTX 580, y en la otra teníamos una NVIDIA Tesla C2075. Respecto a la CPU, en una teníamos un Intel(R) Xeon CPU X5650 @2.67GHz con 23 núcleos, y en el otro ordenador un procesador Intel Core i5-4690 CPU @3.50 GHz de 4 núcleos.

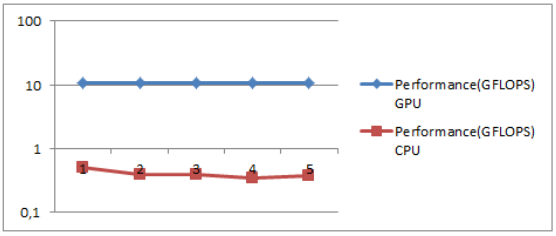
Se han realizado 2 tandas de simulaciones. En la primera tanda se han realizado simulaciones de la 1 a la 10, haciendo 30 simulaciones de cada tamaño de matriz y haciendo una media del tiempo entre ellas. Posteriormente, se han realizado las simulaciones desde la 11 a la 20. Como los tamaños eran mucho mayores y el tiempo en simular era demasiado largo, de esta solo se han hecho 3 simulaciones para cada tamaño de matriz (por lo tanto los datos son menos fiables) y se ha hecho la media del tiempo entre las 3 simulaciones.

Primeramente se han realizado simulaciones en el ordenador con tarjeta gráfica Tesla. Se ha empezado a simular para matrices pequeñas, y se ha ido aumentando el tamaño para comprobar la eficiencia entre una GPU y una CPU.

SIMULACIONES CUDA TESLA C2075

	Matrix A		Matrix B		Matrix C		Performance(GFLOPS)		Efficiency
Simulacion	Height	Width	Height	Width	Heigth	Width	GPU	CPU	
1	640	320	320	640	640	640	10,58	0,5	21,27
2	1280	640	640	1280	1280	1280	10,75	0,39	27,5
3	1920	960	960	1920	1920	1920	10,79	0,39	27,85
4	2560	1280	1280	2560	2560	2560	10,81	0,35	30,91
5	3200	1600	1600	3200	3200	3200	10,83	0,37	29,23
6	3840	1920	1920	3840	3840	3840 x	x	x	

Tabla 9.1. Resultados de simulaciones CUDA en Tesla con altura doble que anchura (Elaboración propia)



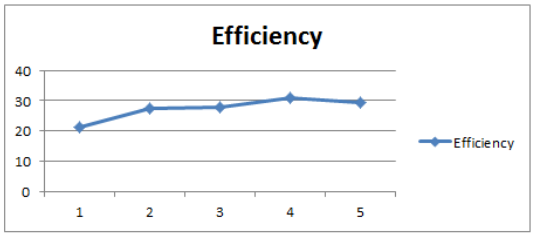


Figura 9.1(a) Performance en GFLOPS de simulaciones CUDA en Tesla con altura doble que anchura. Figura 9.1(b) Eficiencia de simulaciones CUDA en Tesla con altura doble que anchura. (Elaboración propia)

En este primer cuadro observamos que la altura de la matriz A es el doble que su anchura, y al revés para la matriz B. Se han intentado hacer un total de 20 simulaciones, sin embargo, en la sexta simulación el tiempo de ejecución era ya demasiado grande, lo que hacía que el ordenador no permitiera que se ejecutaran más simulaciones. En el apartado "*Performance*" incluimos los GFLOPS tanto de la CPU de la GPU, y en el apartado "*Efficiency*" una ratio entre estos GFLOPS. Podemos observar como en GPU cuanto más grandes son las matrices los GFLOPS aumentan, ya que aunque el tiempo de ejecución aumente, el número de operaciones aumenta en un orden mayor. Sin embargo, en la CPU ocurre lo contrario. Esto hace que los GFLOPS se reduzcan y que la eficiencia vaya aumentando conforme aumenta el tamaño de las matrices. Además, se puede observar como desde el inicio con GPU alcanzamos los 10 GFLOPS, mientras que en CPU lo máximo que llegamos a alcanzar son 0,5 GFLOPS. La diferencia es considerable.

En la gráfica podemos observar mejor la diferencia entre la GPU y la CPU. Mientras que en la GPU se mantiene o incluso aumenta ligeramente, en la CPU los valores caen, y esto hace que la eficiencia aumente.

SIMULACIONES CUDA TESLA C2075									
	Matrix A		Matrix B		Matrix C		Performance(GFLOPS)		Efficiency
Simulacion	Height	Width	Height	Width	Heigth	Width	GPU	CPU	
1	320	320	320	320	320	320	9,51	0,53	17,95
2	640	640	640	640	640	640	10,63	0,43	24,52
3	960	960	960	960	960	960	10,7	0,42	25,23
4	1280	1280	1280	1280	1280	1280	10,76	0,37	28,69
5	1600	1600	1600	1600	1600	1600	10,8	0,38	28,54
6	1920	1920	1920	1920	1920	1920	10,8	0,37	29,27
7	2240	2240	2240	2240	2240	2240	10,84	0,38	28,84
8	2560	2560	2560	2560	2560	2560	10,82	0,19	58
9	2880	2880	2880	2880	2880	2880 x	x	x	

Tabla 9.2. Resultados de simulaciones CUDA en Tesla con altura igual que anchura. (Elaboración propia)

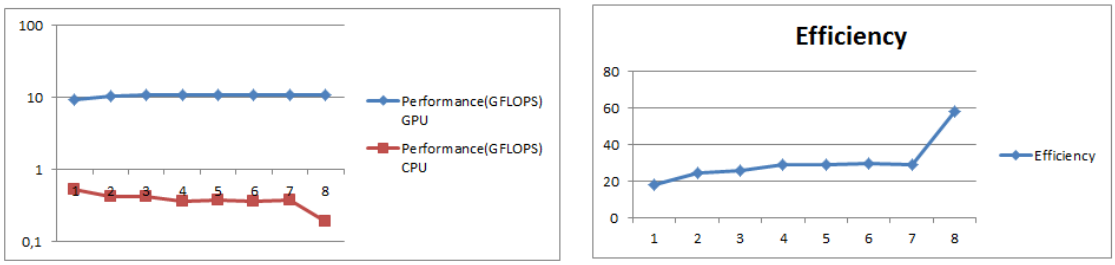


Figura 9.2(a) Performance en GFLOPS de simulaciones CUDA en Tesla con altura igual que anchura. Figura 9.2(b) Eficiencia de simulaciones CUDA en Tesla con altura igual que anchura. (Elaboración propia)

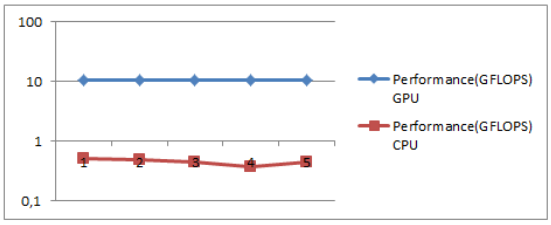
En esta segunda tanda de simulaciones se ha realizado la operación de multiplicación de matrices entre matrices de igual tamaño. Esto hace que sea bastante más rápido que la multiplicación de matrices de distintos tamaños. Además, también podemos observar cómo llega un momento en el que el ordenador deja de responder. En este caso es en la iteración número 9, sin embargo, se puede observar que el tamaño de las matrices inicialmente es más pequeño, por lo tanto resulta normal esperar que sean más iteraciones las que el ordenador pueda soportar. En este cuadro podemos observar dos fenómenos que van a ocurrir mucho a lo largo de las simulaciones, en lugar, una reducción de la eficiencia en la primera iteración. Esto

es debido a que en la CPU los GFLOPS son mayores. Esto lo veremos más detalladamente en futuras gráficas. El otro fenómeno ocurre en la iteración número 8, y en los múltiplos de ésta (como la 16). En esta iteración el rendimiento de la CPU decae radicalmente, haciendo que la eficiencia llegue a triplicarse. No se ha encontrado un motivo de peso por el que pueda ocurrir este fenómeno, pero lo más probable es que sea debido al sistema de colocación de los datos en memoria, que hacen que en la CPU vaya mucho más lento. Podemos observar de nuevo como las realizaciones en GPU oscilan en torno a 10 GFLOPS y en CPU a 0,4 GFLOPS, lo que nos deja una eficiencia media de 25.

En la gráfica podemos observar con más detalle el efecto ocasionado en las simulaciones 1 y 8. En la gráfica de la derecha se puede observar cómo al principio en la CPU el rendimiento aumenta y en la simulación 8 el rendimiento decae (realizado en color rojo). Y como la eficiencia, en el cuadro de la derecha, inicialmente es inferior, se estabiliza en torno a 25 unidades y finalmente se dispara en la octava iteración.

SIMULACIONES CUDA TESLA C2075									
	Matrix A		Matrix B		Matrix C		Performance(GFLOPS)		Efficiency
Simulacion	Height	Width	Height	Width	Heigth	Width	GPU	CPU	
1	480	320	320	800	480	800	10,62	0,51	20,83
2	960	640	640	1600	960	1600	10,72	0,5	21,44
3	1440	960	960	2400	1440	2400	10,79	0,45	24,21
4	1920	1280	1280	3200	1920	3200	10,82	0,37	29,25
5	2400	1600	1600	4000	2400	4000	10,83	0,44	24,37
6	2880	1920	1920	4800	2880	4800	x	x	x

Tabla 9.3. Resultados de simulaciones CUDA en Tesla con todos los valores dispares (Elaboración propia)



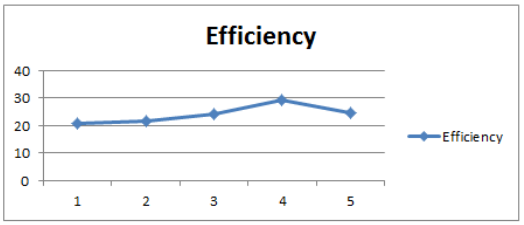


Figura 9.3(a) Performance en GFLOPS de simulaciones CUDA en Tesla con todos los valores dispares. Figura 9.3(b) Eficiencia de simulaciones CUDA en Tesla con todos los valores dispares. (Elaboración propia)

En esta última tanda de simulaciones con tarjeta Tesla en CUDA podemos observar un rendimiento muy similar al de las dos anteriores. En ésta solo se llegan a hacer 5 simulaciones exitosas antes de que el ordenador no pueda continuar debido a la capacidad de computación.

Podemos observar un comportamiento homogéneo a lo largo de toda la tanda de simulaciones, en la que el rendimiento en GPU es de unos 10 GFLOPS y en CPU oscila entre 0,5 y 0,4 GFLOPS.

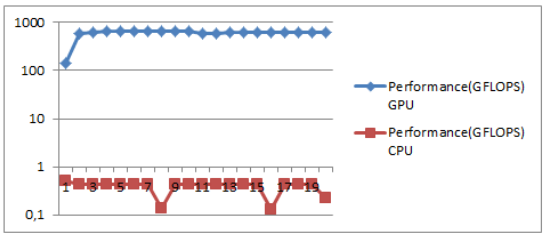
Aunque la eficiencia es ligeramente inferior, no podemos concluir que la diferencia de los tamaños en las matrices influya trascendentalmente en el tiempo de simulación, y por consecuencia, en los GFLOPS.

A continuación se muestran los resultados de las simulaciones de multiplicación de matrices con el código de la librería cuBLAS hechas con una tarjeta tesla C2075.

SIMULACIONES CUBLAS TESLA C2075

	Matrix A		Matrix B		Matrix C		Performance(GFLOPS)		Efficiency
Simulacion	Height	Width	Height	Width	Heigth	Width	GPU	CPU	
1	640	320	320	640	640	640	139,73	0,5	278,11
2	1280	640	640	1280	1280	1280	598,96	0,43	1393,84
3	1920	960	960	1920	1920	1920	636,01	0,44	1458,05
4	2560	1280	1280	2560	2560	2560	648,97	0,43	1496,71
5	3200	1600	1600	3200	3200	3200	659,73	0,43	1519,53
6	3840	1920	1920	3840	3840	3840	658,17	0,43	1518,53
7	4480	2240	2240	4480	4480	4480	657,5	0,44	1505,25
8	5120	2560	2560	5120	5120	5120	656,66	0,14	4560,9
9	5760	2880	2880	5760	5760	5760	655,09	0,44	1484,48
10	6400	3200	3200	6400	6400	6400	651,47	0,44	1489,26
11	7040	3520	3520	7040	7040	7040	579,05	0,44	1316,02
12	7680	3840	3840	7680	7680	7680	588,21	0,43	1367,93
13	8320	4160	4160	8320	8320	8320	616,45	0,44	1401,02
14	8960	4480	4480	8960	8960	8960	623,18	0,44	1416,31
15	9600	4800	4800	9600	9600	9600	624,11	0,44	1418,43
16	10240	5120	5120	10240	10240	10240	621,54	0,13	4781,06
17	10880	5440	5440	10880	10880	10880	626,29	0,44	1423,34
18	11520	5760	5760	11520	11520	11520	627,13	0,44	1425,29
19	12160	6080	6080	12160	12160	12160	627,06	0,44	1425,13
20	12800	6400	6400	12800	12800	12800	627,86	0,22	2853,9

Tabla 9.4. Resultados de simulaciones cuBLAS en Tesla con altura doble que anchura (Elaboración propia)



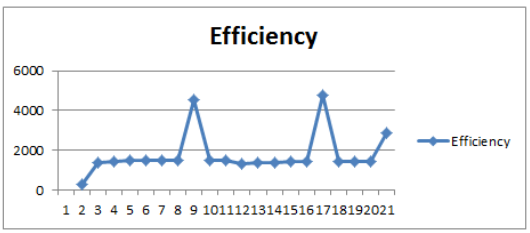


Figura 9.4(a) Performance en GFLOPS de simulaciones cuBLAS en Tesla con altura doble que anchura. Figura 9.4(b) Eficiencia de simulaciones cuBLAS en Tesla con altura doble que anchura. (Elaboración propia)

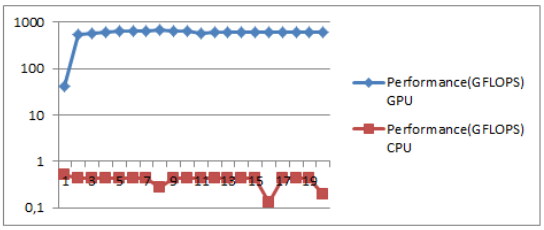
A diferencia de en CUDA, en estas simulaciones se pueden observar diferencias enormes en la eficiencia de las realizaciones. Mientras que utilizando CUDA se llegaban a las decenas de GFLOPS, en cuBLAS fácilmente se alcanzan unos cuantos centenares. Esto es debido a que la librería cuBLAS está preparada de por sí para la realización de operaciones matemáticas ya que es en lo que se especializa. Por lo tanto, los accesos a memoria y la forma en que los datos se guardan en ésta se realizan de la forma más eficiente posible. Además, la selección del tamaño de *grid* y del bloque se hace de forma perfecta, de modo que no hay forma de que se desperdicie espacio. Por ello y por todo lo anterior podemos ver una mejora en eficiencia tanto con respecto a CPU como con respecto a CUDA.

En este cuadro podemos observar también lo comentado anteriormente, unos picos de eficiencia en las simulaciones 8 y 16, además de un ligero pico en la simulación 20. Además, se puede observar como la realización de la primera simulación no alcanza la media de las demás. Se podría pensar que es porque el tamaño de las matrices empieza a ser significante, pero se han hecho distintas pruebas y se ha comprobado que es debido a que en el resto de simulaciones hay datos que se mantienen en la caché, lo que hace que sea más rápido acceder a ellos y que por lo tanto la primera vez que se accede sea mucho más lenta que el resto de veces.

SIMULACIONES CUBLAS TESLA C2075

	Matrix A		Matrix B		Matrix C		Performance(GFLOPS)		Efficiency
Simulacion	Height	Width	Height	Width	Heigth	Width	GPU	CPU	
1	320	320	320	320	320	320	42,78	0,51	83,36
2	640	640	640	640	640	619,33	528,62	0,43	1223,33
3	960	960	960	960	960	960	587,62	0,44	1350,16
4	1280	1280	1280	1280	1280	1280	619,33	0,44	1402,97
5	1600	1600	1600	1600	1600	1600	639,99	0,44	1467,39
6	1920	1920	1920	1920	1920	1920	652,21	0,44	1487,1
7	2240	2240	2240	2240	2240	2240	656,17	0,44	1498,56
8	2560	2560	2560	2560	2560	2560	664,46	0,27	2488,53
9	2880	2880	2880	2880	2880	2880	660,76	0,44	1499,93
10	3200	3200	3200	3200	3200	3200	660,06	0,44	1501,73
11	3520	3520	3520	3520	3520	3520	559,05	0,44	1276,52
12	3840	3840	3840	3840	3840	3840	598,21	0,43	1380,12
13	4160	4160	4160	4160	4160	4160	613,37	0,44	1396,27
14	4480	4480	4480	4480	4480	4480	613,71	0,44	1393,15
15	4800	4800	4800	4800	4800	4800	614,11	0,44	1396,83
16	5120	5120	5120	5120	5120	5120	615,34	0,13	4742,84
17	5440	5440	5440	5440	5440	5440	616,9	0,44	1399,55
18	5760	5760	5760	5760	5760	5760	617,73	0,44	1401,83
19	6080	6080	6080	6080	6080	6080	617,96	0,44	1403,36
20	6400	6400	6400	6400	6400	6400	617,03	0,2	3142,65

Tabla 9.5. Resultados de simulaciones cuBLAS en Tesla con altura igual que anchura (Elaboración propia)



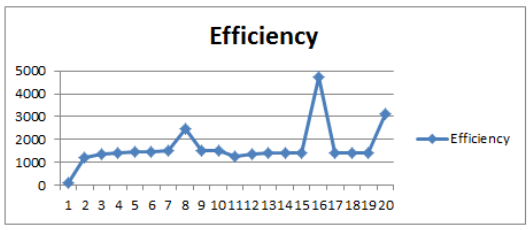
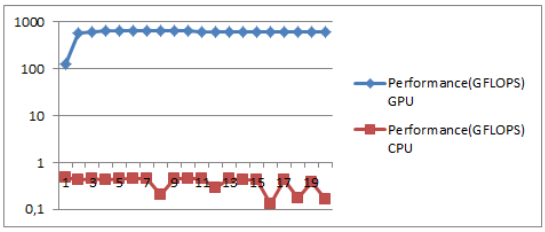


Figura 9.5(a) Performance en GFLOPS de simulaciones cuBLAS en Tesla con altura igual que anchura. Figura 9.5(b) Eficiencia de simulaciones cuBLAS en Tesla con altura igual que anchura. (Elaboración propia)

En la tabla 9.5 y sus respectivas figuras 9.5(a y b) podemos observar las simulaciones realizadas sobra la tarjeta Tesla con matrices cuadradas. No hay ningún dato reseñable que destacar aparte de los dichos en las tablas y figuras 9.4. Se podría destacar levemente el bajo rendimiento que se obtiene en la primera simulación, esta vez pudiendo ser no sólo de la caché, sino también del pequeño tamaño de las matrices.

SIMULACIONES CUBLAS TESLA C2075									
	Matrix A		Matrix B		Matrix C		Performance(GFLOPS)		Efficiency
Simulacion	Height	Width	Height	Width	Heigth	Width	GPU	CPU	
1	480	800	800	320	480	320	124,54	0,49	253,64
2	960	1600	1600	640	960	640	579,96	0,43	1345,89
3	1440	2400	2400	960	1440	960	616,34	0,44	1410,99
4	1920	3200	3200	1280	1920	1280	652,6	0,43	1500,91
5	2400	4000	4000	1600	2400	1600	656,04	0,44	1506,04
6	2880	4800	4800	1920	2880	1920	663,09	0,44	1516,57
7	3360	5600	5600	2240	3360	2240	651,06	0,44	1487,45
8	3840	6400	6400	2560	3840	2560	654,59	0,21	3160,73
9	4320	7200	7200	2880	4320	2880	647,8	0,44	1471,16
10	4800	8000	8000	3200	4800	3200	647,42	0,44	1470,41
11	5280	8800	8800	3520	5280	3520	596,23	0,44	1361,97
12	5760	9600	9600	3840	5760	3840	618,48	0,29	2124,4
13	6240	10400	10400	4160	6240	4160	620,26	0,44	1416,11
14	6720	11200	11200	4480	6720	4480	616,56	0,43	1435,61
15	7200	12000	12000	4800	7200	4800	620,03	0,43	1439,5
16	7680	12800	12800	5120	7680	5120	621,02	0,13	4903,68
17	8160	13600	13600	5440	8160	5440	620,96	0,42	1465,87
18	8640	14400	14400	5760	8640	5760	619,47	0,17	3645,14
19	9120	15200	15200	6080	9120	6080	623,65	0,39	1608,44
20	9600	16000	16000	6400	9600	6400	626,66	0,16	4004,88

Tabla 9.6. Resultados de simulaciones cuBLAS en Tesla con valores dispares (Elaboración propia)



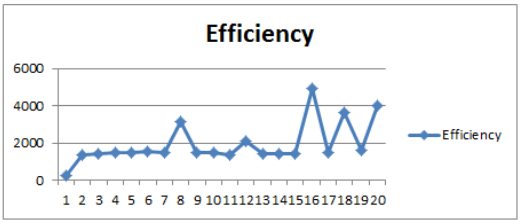


Figura 9.6(a) Performance en GFLOPS de simulaciones cuBLAS en Tesla con todos los valores dispares. Figura 9.6(b) Eficiencia de simulaciones cuBLAS en Tesla con todos los valores dispares. (Elaboración propia)

En la tabla 9.6 y sus respectivas figuras 9.6(a y b) podemos observar las simulaciones realizadas sobre la tarjeta Tesla con valores de anchura y altura de las matrices completamente distintas. Aunque se observa una ligera caída de rendimiento en las primeras fases de la simulación, en las últimas se puede observar que de nuevo se estabilizan los valores.

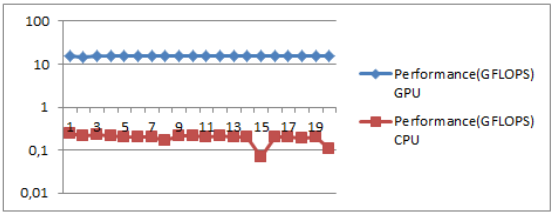
Cabe la posibilidad de que para tamaños pequeños de la matriz sí que influyan los valores dispares de altura y anchura, ya que la agrupación de 32 en 32 de los *warps* pueda influir porque haya algún *warp* vacío o que no encaje correctamente, lo que haga que se desperdicie espacio. Sin embargo, al aumentar el tamaño de las matrices las dimensiones van aumentando de tamaño, y pueden hacer que éstas sean múltiplos de los tamaños de almacenaje.

Un valor aislado pero que se repite en todas las simulaciones realizadas es la simulación número 18. Si observamos en la tabla 9.6 podemos ver un descenso drástico del rendimiento de la CPU en esta simulación, que no encaja con ninguno de los patrones establecidos. Aunque se está intentando buscar algún motivo de este fenómeno, aún no se ha llegado a una conclusión. Por lo tanto pasaremos simplemente a destacarlo como dato para futuras investigaciones.

Pasemos ahora al estudio con la tarjeta gráfica GTX 580. En estas simulaciones se utilizaron 3 tarjetas gráficas NVIDIA GeForce GTX 580 en comparación con un procesador Intel(R) Xeon CPU X5650 @2.67GHz x 23 núcleos.

SIMULACIONES CUBLAS GTX 580									
	Matrix A		Matrix B		Matrix C		Performance(GFLOPS)		Efficiency
Simulacion	Height	Width	Height	Width	Heigth	Width	GPU	CPU	
1	640	320	320	640	640	640	15,3	0,25	61,3
2	1280	640	640	1280	1280	1280	14,74	0,22	67,48
3	1920	960	960	1920	1920	1920	15,05	0,23	66,79
4	2560	1280	1280	2560	2560	2560	15,16	0,21	73,34
5	3200	1600	1600	3200	3200	3200	15,21	0,2	75,44
6	3840	1920	1920	3840	3840	3840	15,25	0,2	75,89
7	4480	2240	2240	4480	4480	4480	15,27	0,2	75,66
8	5120	2560	2560	5120	5120	5120	15,29	0,17	90,36
9	5760	2880	2880	5760	5760	5760	15,31	0,21	74,52
10	6400	3200	3200	6400	6400	6400	15,31	0,21	74,7
11	7040	3520	3520	7040	7040	7040	15,33	0,2	76,29
12	7680	3840	3840	7680	7680	7680	15,31	0,21	73,97
13	8320	4160	4160	8320	8320	8320	15,34	0,2	76,3
14	8960	4480	4480	8960	8960	8960	15,35	0,2	77,36
15	9600	4800	4800	9600	9600	9600	15,35	0,07	217,61
16	10240	5120	5120	10240	10240	10240	15,36	0,2	74,93
17	10880	5440	5440	10880	10880	10880	15,36	0,2	74,93
18	11520	5760	5760	11520	11520	11520	15,36	0,19	79,72
19	12160	6080	6080	12160	12160	12160	15,36	0,2	76,55
20	12800	6400	6400	12800	12800	12800	15,38	0,11	139,72

Tabla 9.7. Resultados de simulaciones CUDA en GTX con altura doble que anchura (Elaboración propia)



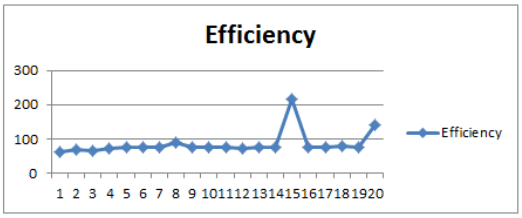


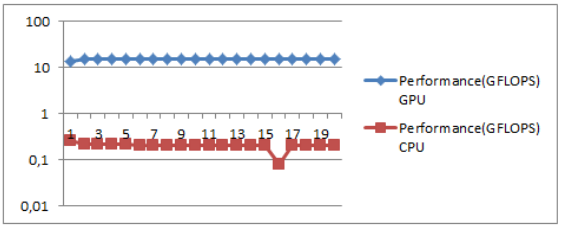
Figura 9.7(a) Performance en GFLOPS de simulaciones CUDA en GTX con altura doble que anchura. Figura 9.7(b) Eficiencia de simulaciones CUDA en GTX con altura doble que anchura. (Elaboración propia)

Podemos observar, en principio, que los resultados obtenidos con estas tarjetas gráficas son mejores que las anteriores. Mientras que en las anteriores observábamos un rendimiento de en torno a 10 GFLOPS, en este caso el rendimiento asciende a 15 GFLOPS, es decir, un 50% más. Además, la CPU obtiene un peor rendimiento, pasando de 0,4 GFLOPS a 0,2 GFLOPS, algo que deberemos tener en cuenta a lo largo de las simulaciones ya que se verá afectada la eficiencia entre ambas de forma considerable.

Otro aspecto importante es que con esta CPU y tarjetas gráficas el programa sí que nos permite terminar las 20 simulaciones (recordamos en la tabla 9.1 que llegaba un momento en el que el programa no dejaba continuar la simulación). Lo que nos permite ver una disparidad en la simulación número 15 (figura 9.7(b)), en la que las matrices valen 9.600 x 4.800. De nuevo, parece que debe ser por el tamaño de las matrices y la forma de almacenarlas.

SIMULACIONES CUBLAS GTX 580									
	Matrix A		Matrix B		Matrix C		Performance(GFLOPS)		Efficiency
Simulacion	Height	Width	Height	Width	Heigth	Width	GPU	CPU	
1	320	320	320	320	320	320	13,61	0,26	51,53
2	640	640	640	640	640	619,33	15,34	0,22	70,13
3	960	960	960	960	960	960	15,17	0,22	69,01
4	1280	1280	1280	1280	1280	1280	15,38	0,22	70,69
5	1600	1600	1600	1600	1600	1600	15,32	0,22	69
6	1920	1920	1920	1920	1920	1920	15,39	0,21	74,74
7	2240	2240	2240	2240	2240	2240	15,36	0,2	75,11
8	2560	2560	2560	2560	2560	2560	15,4	0,2	75,59
9	2880	2880	2880	2880	2880	2880	15,37	0,21	74,38
10	3200	3200	3200	3200	3200	3200	15,4	0,2	75,67
11	3520	3520	3520	3520	3520	3520	15,35	0,2	76,66
12	3840	3840	3840	3840	3840	3840	15,37	0,2	76,78
13	4160	4160	4160	4160	4160	4160	15,33	0,2	75,49
14	4480	4480	4480	4480	4480	4480	15,37	0,21	74,66
15	4800	4800	4800	4800	4800	4800	15,37	0,2	76,13
16	5120	5120	5120	5120	5120	5120	15,38	0,08	195,93
17	5440	5440	5440	5440	5440	5440	15,38	0,2	75,24
18	5760	5760	5760	5760	5760	5760	15,38	0,2	76,28
19	6080	6080	6080	6080	6080	6080	15,38	0,2	76,51
20	6400	6400	6400	6400	6400	6400	15,37	0,2	76,83

Tabla 9.8. Resultados de simulaciones CUDA en GTX con altura igual que anchura (Elaboración propia)



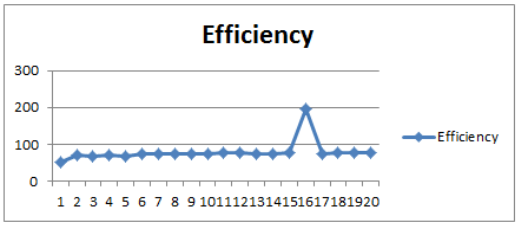


Figura 9.8(a) Performance en GFLOPS de simulaciones CUDA en GTX con altura igual que anchura.
Figura 9.8(b) Eficiencia de simulaciones CUDA en GTX con altura igual que anchura.

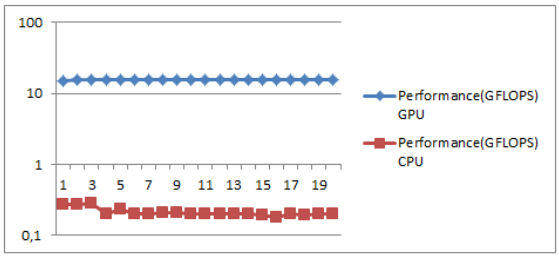
En la tabla 9.8 podemos observar los datos de las simulaciones cuando nuestras matrices tienen dimensiones cuadradas. Podemos observar un comportamiento similar al de la tabla anterior, sin esa extraña bajada de rendimiento de la CPU en la simulación número 15. Sin embargo, esta vez ocurre en la simulación número 16, de nuevo sospechamos que es por lo mismo.

Podemos observar una ligera bajada de GFLOPS en la primera simulación, posiblemente debido a la caché. Esta bajada no es tan significativa como ocurría en cuBLAS, seguramente porque cuBLAS es una librería centrada en la matemática que aprovechaba las cachés de una forma más eficiente, lo que hacía que el cambio se notara aún más.

Por último, otro dato a destacar es que de nuevo se nos permite alcanzar la simulación número 20 sin ningún requerimiento de tiempo, aunque debido a la CPU el tiempo de simulación haya sido mayor. Esto se debe a la configuración de cada uno de los Nsight, que puede ser cambiada a disposición del usuario.

SIMULACIONES CUBLAS GTX 580									
	Matrix A		Matrix B		Matrix C		Performance(GFLOPS)		Efficiency
Simulacion	Height	Width	Height	Width	Heigth	Width	GPU	CPU	
1	480	800	800	320	480	320	14,94	0,27	56,19
2	960	1600	1600	640	960	640	15,33	0,27	57,32
3	1440	2400	2400	960	1440	960	15,38	0,28	55,6
4	1920	3200	3200	1280	1920	1280	15,39	0,2	78,55
5	2400	4000	4000	1600	2400	1600	15,39	0,23	66,31
6	2880	4800	4800	1920	2880	1920	15,39	0,2	77,37
7	3360	5600	5600	2240	3360	2240	15,39	0,2	77,01
8	3840	6400	6400	2560	3840	2560	15,39	0,21	75,03
9	4320	7200	7200	2880	4320	2880	15,39	0,21	75,17
10	4800	8000	8000	3200	4800	3200	15,4	0,2	77,14
11	5280	8800	8800	3520	5280	3520	15,38	0,2	76,87
12	5760	9600	9600	3840	5760	3840	15,38	0,2	76,38
13	6240	10400	10400	4160	6240	4160	15,36	0,2	75,98
14	6720	11200	11200	4480	6720	4480	15,38	0,2	75,3
15	7200	12000	12000	4800	7200	4800	15,39	0,19	83,03
16	7680	12800	12800	5120	7680	5120	15,38	0,18	86,44
17	8160	13600	13600	5440	8160	5440	15,38	0,2	76,46
18	8640	14400	14400	5760	8640	5760	15,39	0,19	79,29
19	9120	15200	15200	6080	9120	6080	15,39	0,2	77,47
20	9600	16000	16000	6400	9600	6400	15,38	0,2	76,68

Figura 9.9(a) Rendimiento en GFLOPS de simulaciones CUDA en GTX con todos los valores dispares. (Elaboración propia)



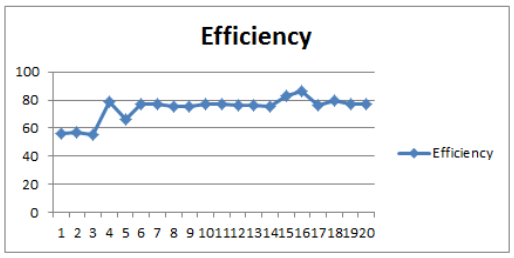


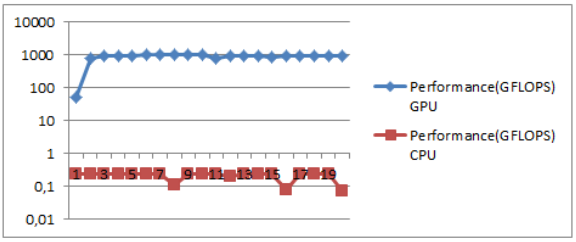
Figura 9.9(a) Performance en GFLOPS de simulaciones CUDA en GTX con todos los valores dispares. Figura 9.9(b) Eficiencia de simulaciones CUDA en GTX con todos los valores dispares. (Elaboración propia)

La tabla 9.9 y las figuras 9.9(a y b) corresponden a la simulación con la tarjeta gráfica GeForce GTX 580 con valores de matrices dispares. En este caso podemos ver un aumento del rendimiento en CPU en los primeros valores de las matrices, es decir, para tamaños pequeños. Rendimiento que posteriormente se diluye para volver de nuevo a los 0,2 GFLOPS. Esto se puede observar perfectamente en la figura 9.9(b), ya que vemos como la eficiencia se sitúa en principio en unas 55 unidades, pasando a 80 en la cuarta simulación.

Por lo demás, vemos de nuevo valores similares respecto a las otras dos simulaciones que se han realizado anteriormente.

SIMULACIONES CUBLAS GTX 580									
	Matrix A		Matrix B		Matrix C		Performance(GFLOPS)		Efficiency
Simulacion	Height	Width	Height	Width	Heigth	Width	GPU	CPU	
1	640	320	320	640	640	640	51,62	0,24	213,62
2	1280	640	640	1280	1280	1280	821,52	0,23	3650,89
3	1920	960	960	1920	1920	1920	903,5	0,23	3925,48
4	2560	1280	1280	2560	2560	2560	976,57	0,23	4219,14
5	3200	1600	1600	3200	3200	3200	981,2	0,23	4245,11
6	3840	1920	1920	3840	3840	3840	1002,68	0,23	4271,73
7	4480	2240	2240	4480	4480	4480	1007,01	0,23	4292,6
8	5120	2560	2560	5120	5120	5120	1013,64	0,11	8886,16
9	5760	2880	2880	5760	5760	5760	1012,3	0,23	4316,44
10	6400	3200	3200	6400	6400	6400	1013,92	0,24	4283,83
11	7040	3520	3520	7040	7040	7040	805,25	0,23	3434,64
12	7680	3840	3840	7680	7680	7680	918,58	0,2	4658,98
13	8320	4160	4160	8320	8320	8320	923,74	0,23	3934,96
14	8960	4480	4480	8960	8960	8960	930,49	0,23	3974,19
15	9600	4800	4800	9600	9600	9600	887,31	0,23	3790,57
16	10240	5120	5120	10240	10240	10240	922,87	0,08	11880,81
17	10880	5440	5440	10880	10880	10880	921,7	0,24	3917,26
18	11520	5760	5760	11520	11520	11520	930,15	0,24	3938,36
19	12160	6080	6080	12160	12160	12160	933,46	0,24	3917,29
20	12800	6400	6400	12800	12800	12800	938,46	0,07	12629,23

Tabla 9.10. Resultados de simulaciones cuBLAS en GTX con altura doble que anchura (Elaboración propia)



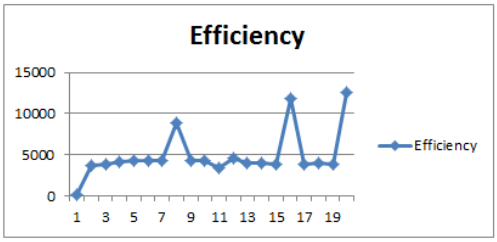


Figura 9.10(a) Performance en GFLOPS de simulaciones cuBLAS en GTX con altura doble que anchura. Figura 9.10(b) Eficiencia de simulaciones cuBLAS en GTX con altura doble que anchura. (Elaboración propia)

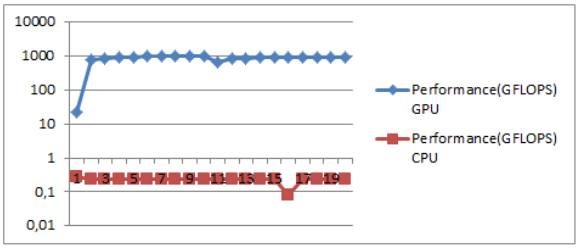
En la tabla 9.10 y figuras 9.10(a y b) podemos observar la tanda de simulaciones realizadas para la multiplicación de matrices en cuBLAS con una tarjeta gráfica GeForce GTX 580 donde la altura es el doble de la anchura de las matrices.

Al igual que pudimos observar en las tablas 9.7, 9.8 y 9.9 donde el rendimiento aumentaba un 50%, en este caso el rendimiento aumenta también un 50%. Mientras que con la tarjeta gráfica Tesla obteníamos rendimientos de en torno a los 600 GFLOPS, con la GTX alcanzamos fácilmente los 900 GFLOPS. Algo que, si añadimos a la reducción de rendimiento en CPU, hace que la eficiencia en cuBLAS se dispare. Mientras que en Tesla obteníamos una eficiencia de 1.400 unidades, aquí se alcanzan las 4.000 de forma sencilla.

Además, podemos observar las bajadas de rendimiento de la CPU características de las simulaciones 8, 16 y 20.Además, también se puede observar la caída de rendimiento en GPU en la primera simulación, debido a la falta de recursos guardados en caché.

SIMULACIONES CUBLAS GTX 580									
	Matrix A		Matrix B		Matrix C		Performance(GFLOPS)		Efficiency
Simulacion	Height	Width	Height	Width	Heigth	Width	GPU	CPU	
1	320	320	320	320	320	320	21,94	0,27	79,81
2	640	640	640	640	640	619,33	756,13	0,23	3318,11
3	960	960	960	960	960	960	866,53	0,23	3796,1
4	1280	1280	1280	1280	1280	1280	943,06	0,23	4171,61
5	1600	1600	1600	1600	1600	1600	957,4	0,23	4078,55
6	1920	1920	1920	1920	1920	1920	984,38	0,23	4211,68
7	2240	2240	2240	2240	2240	2240	993,43	0,23	4262,93
8	2560	2560	2560	2560	2560	2560	1000,79	0,23	4262,65
9	2880	2880	2880	2880	2880	2880	1005,9	0,23	4288,16
10	3200	3200	3200	3200	3200	3200	1009,04	0,23	4364,52
11	3520	3520	3520	3520	3520	3520	679,03	0,24	2866,09
12	3840	3840	3840	3840	3840	3840	843,85	0,24	3587,36
13	4160	4160	4160	4160	4160	4160	885,91	0,23	3770,33
14	4480	4480	4480	4480	4480	4480	896,41	0,23	3823,64
15	4800	4800	4800	4800	4800	4800	905,66	0,24	3804,69
16	5120	5120	5120	5120	5120	5120	911,65	0,08	11643,02
17	5440	5440	5440	5440	5440	5440	915,64	0,24	3838,14
18	5760	5760	5760	5760	5760	5760	920,14	0,23	3934
19	6080	6080	6080	6080	6080	6080	926,75	0,24	3905,02
20	6400	6400	6400	6400	6400	6400	931,52	0,24	3952,96

Tabla 9.11. Resultados de simulaciones cuBLAS en GTX con altura igual que anchura (Elaboración propia)



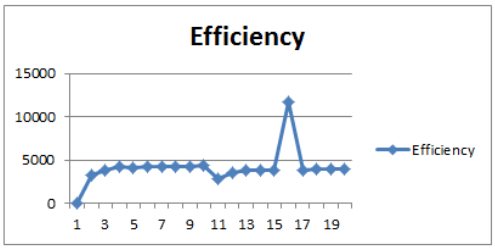


Figura 9.11(a) Performance en GFLOPS de simulaciones cuBLAS en GTX con altura igual que anchura. Figura 9.11(b) Eficiencia de simulaciones cuBLAS en GTX con altura igual que anchura. (Elaboración propia)

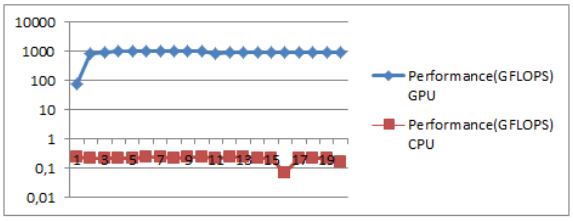
En la tabla 9.11 y figuras 9.11(a y b) podemos observar la tanda de simulaciones realizadas para la multiplicación de matrices en cuBLAS con una tarjeta gráfica GeForce GTX 580 donde la altura es el igual que la anchura de las matrices.

Podemos observar un comportamiento muy similar al de la tabla 9.10, aunque esta vez sin los picos en 8 y 20, pero sí el pico de 16. También se puede observar la bajada de eficiencia de la primera simulación.

Algo importante y que no se ha tenido en cuenta hasta ahora (porque en la gráfica 9.11(b) es donde mejor se ve) es la bajada de eficiencia que ha habido en algunas gráficas en la simulación 11. Esto es debido a, como se dijo al principio de este apartado, las simulaciones se han realizado en dos tandas, una de la 1 a la 10, y otra de la 11 a la 20. Por ello la bajada de eficiencia de la 11 es debido a la caché, al igual que en la 1. Sin embargo, en la caché, al tener tamaños de matriz inferiores, los efectos son más notables.

SIMULACIONES CUBLAS GTX 580									
	Matrix A		Matrix B		Matrix C		Performance(GFLOPS)		Efficiency
Simulacion	Height	Width	Height	Width	Heigth	Width	GPU	CPU	
1	480	800	800	320	480	320	75,55	0,25	305,12
2	960	1600	1600	640	960	640	862,65	0,23	3806,12
3	1440	2400	2400	960	1440	960	920,53	0,23	4060,92
4	1920	3200	3200	1280	1920	1280	980,96	0,23	4250,24
5	2400	4000	4000	1600	2400	1600	984,39	0,23	4210,45
6	2880	4800	4800	1920	2880	1920	997,85	0,24	4183,22
7	3360	5600	5600	2240	3360	2240	999,04	0,24	4249,13
8	3840	6400	6400	2560	3840	2560	1008,07	0,23	4303,49
9	4320	7200	7200	2880	4320	2880	1004,7	0,24	4237,04
10	4800	8000	8000	3200	4800	3200	1012,15	0,24	4249,79
11	5280	8800	8800	3520	5280	3520	833,61	0,23	3563,83
12	5760	9600	9600	3840	5760	3840	928,71	0,24	3899,43
13	6240	10400	10400	4160	6240	4160	932,51	0,24	3952,78
14	6720	11200	11200	4480	6720	4480	939,81	0,23	4023,79
15	7200	12000	12000	4800	7200	4800	941,74	0,23	4143,67
16	7680	12800	12800	5120	7680	5120	950,34	0,07	13790,15
17	8160	13600	13600	5440	8160	5440	944,8	0,23	4129,17
18	8640	14400	14400	5760	8640	5760	955,28	0,23	4159,47
19	9120	15200	15200	6080	9120	6080	952,31	0,23	4190,88
20	9600	16000	16000	6400	9600	6400	962,89	0,16	6098,78

Tabla 9.12. Resultados de simulaciones cuBLAS en GTX con valores dispares (Elaboración propia)



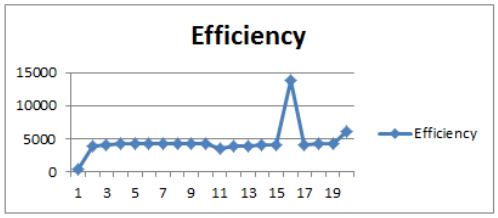


Figura 9.12(a) Performance en GFLOPS de simulaciones cuBLAS en GTX con todos los valores dispares. Figura 9.12(b) Eficiencia de simulaciones cuBLAS en GTX con todos los valores dispares. (Elaboración propia)

Por último, en la tabla 9.12 y figuras 9.12(a y b) podemos observar la tanda de simulaciones realizadas para la multiplicación de matrices en cuBLAS con una tarjeta gráfica GeForce GTX 580 donde la altura y la anchura de ambas matrices son distintas.

Como cabe esperar, los resultados son de nuevo similares al resto de simulaciones.

No hay mucho más que añadir de esta última gráfica. Destacamos los picos de eficiencia de las simulaciones 16 y levemente de la 20 y la bajada en la primera simulación debido a la caché de la 1 y también de la 11.

Como dato, observar que en la simulación 16 se encuentra el pico de eficiencia más alto que ha habido entre GPU y CPU, con un total de 13790,15. Es decir, que la GPU ejecutó el programa 13790,15 veces más rápido que la CPU.

10. Perceptrón multicapa híbrido

Como punto final, se ha tomado como objetivo la implementación de un perceptrón multicapa de forma exitosa. El código puede encontrarse en el apéndice 1.7 *"Perceptrón multicapa híbrido*".

Como ejemplo, se ha desarrollado una aproximación para una función XOR con un perceptrón de dos entradas, una capa oculta con dos neuronas, y una capa de salida con una neurona. Se han realizado ambas etapas, la de testeo y aprendizaje, con éxito.

Comencemos explicando cada uno de los kernels:

template <**int BLOCK_SIZE**> **__global__ void matrixMulCUDA**(**float** *c, **float** *a, **float** *b, **int** wA, **int** wB)
Este kernel es el código que correrá en el *device* para la multiplicación de matrices. Se ha realizado un código de multiplicación de matrices con *templates* (plantillas). La explicación del código ya se hizo anteriormente, para volver a ello, diríjase al apartado 7.5 "*Explotando el paralelismo de las GPUs*".

int matrixmult(**float*** h_C,**float*** h_B,**float*** h_A, **int** hA,**int** wA, **int** wB)

De nuevo, el código ejecutado en la CPU sobre la multiplicación de matrices. Si se quiere volver a la explicación del código diríjase al apartado 7.5 "*Explotando el paralelismo de las GPUs*".

template< **int TILE_DIM**>**__global__ void transpose_dev**(**float** *odata, **float** *idata)

Código en el *device* utilizado para trasponer una matriz. Tenemos dos entradas, una con un puntero a la matriz de entrada y otra con un puntero a la matriz de salida. Se ha definido un bloque con 8 filas y 32 columnas según la plantilla, algo que habría que cambiar en función de la entrada que estemos introduciendo. Se utiliza memoria compartida para acelerar el proceso de transposición ya que la memoria compartida es bastante más rápida. Al utilizar memoria compartida, tenemos que sincronizar los hilos con el comando `syncthreads()`.

La parte importante del código, donde realmente se transpone la matriz, es la siguiente:

```
  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];
  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y +
j];
```

Podemos observar cómo para el dato de salida, el dato que se encuentra en la posición (y+j)*ancho + x corresponde con la posición [x][y+j], mientras que al guardarlo en la memoria compartida, se ha guardado de forma opuesta, es decir [y+j][x]. Lo que hace que nuestra matriz quede traspuesta.

void transpose(**float** *odata, **const float** *idata, **int** dim)

En este kernel queda el código en CPU usado para poder utilizar el kernel `transpose_dev()` de forma correcta. Se inicializan los valores y se reserva la memoria

tanto en el *host* como en el *device*. En esta parte es donde introducimos el valor del número de columnas de la matriz, que quedará guardado dentro de la variable `block`. Este kernel simplemente se basa en un código de búsqueda de errores, reserva de memoria y declaración de variables en CPU.

void transpon(**float** *data_transp, **float** *data, **int** rows, **int** cols)

Mismo código para trasponer una matriz, esta vez únicamente en CPU. Dispone de cuatro entradas, dos para las matrices de entrada y salida y dos valores enteros representando el número de filas y el número de columnas.

De nuevo observamos la trasferencia de datos para trasponerlos, esta vez de forma más sencilla ya que no hay ningún intermediario.

```
void transpon(float *data_transp,float *data, int rows, int
cols){
      for (int i = 0; i < rows; i++)
              for (int j = 0; j < cols; j++)
                  data_transp[(j*rows)+i] = data[(i*cols)+j];
```

void perceptron(**float** *input,**float** *salida_deseada,**float** momentum,**float** rate,**int** Niter, **int** n, **int** iter,**float*** weightsInput, **float*** weightsOutput,**float*** weightsInput_ant,**float*** weightsOutput_ant, **int** patron)

Este es el kernel con el programa principal, donde se desarrolla en sí el perceptrón. Primeramente, tenemos una declaración de variables que vamos a necesitar para ejecutar nuestro perceptrón.

A continuación tenemos unas líneas de código actualizando la entrada que en un principio recibimos.

```
for (int i = 0; i < n*patron; i++) {
             newInput[i+patron] = input[i];
}
for (int i = 0; i < patron; i++) {
 newInput[i] = -1;
}
```

Estas líneas de código introducen un -1 en la primera fila de la matriz de entrada. Esto se debe a algo que se dijo que se explicaría en el apartado 4 "*Perceptrón multicapa*". En este apartado, se dijo que los umbrales eran tratados como pesos donde su activación era igual a 1. Como los umbrales deben ser restados, deberemos multiplicar por -1. Por ello, introducimos una nueva fila en nuestra matriz con todo -1, de forma que con una misma multiplicación de matrices seamos capaces de obtener tanto los pesos como los umbrales. Esto hace que nuestro programa sea mucho más eficiente.

A continuación empezamos el entrenamiento de nuestro perceptrón. Como se va a desarrollar un perceptrón de dos capas, la función de salida correspondiente será igual que la fórmula 2.3.

$$S = F(F(X * W_1) * W_2) \tag{2.3}$$

Donde nuestra matriz de pesos de entrada será `weightsinput` y la de pesos de salida `weightsoutput`.

Por ello, primeramente multiplicamos la entrada por los pesos de entrada, le aplicamos la función de activación, que en este caso es la sigmoidea, a través de la línea de código

```
InputsHiddenTAN[j] = 1/(1+exp(-InputsHidden[j]));
```

volvemos a multiplicarlo por los pesos de salida y le aplicamos de nuevo la función para obtener finalmente la salida.

Finalmente obtenemos un error dependiendo de la salida deseada que hemos introducido en el entrenamiento.

Para una mejor comprensión y análisis, se escriben por pantalla los pesos antes de ser modificados.

A continuación se comprueba el criterio de parada. En este ejemplo realizado, el programa se parará cuando el error mse sea inferior a una determinada tolerancia que viene determinada al principio del programa. Se irán mostrando por pantalla los errores, para observar cómo van decreciendo, hasta que finalmente el programa llegará a una solución.

Respecto a la parte del entrenamiento, se realizará en dos etapas. Una primera etapa donde el error se propagará a la capa oculta, y obtendremos la regla delta para la última capa, y una segunda etapa donde ese error se propaga al principio y obtenemos la regla delta para la capa anterior.

Comenzaremos propagando el error hacia atrás. Para ello, lo primero que hay que hacer es pasar la función de activación hacia atrás, que es lo que se hace en las primeras líneas del código:

```
for(int k=0;k<patron;k++){
d_OutputsHidden[k] =  OutputsHidden[k] * (1 - OutputsHidden[k]);
d_OutputsHidden[k] = d_OutputsHidden[k] * error[k];
}
```

A continuación, multiplicamos la matriz transpuesta de entradas a esta última capa por lo obtenido tras propagar el error hacia atrás, con lo que obtendríamos, si lo multiplicamos por el ratio de aprendizaje, nuestro término delta, con lo que actualizaremos nuestros pesos según la ecuación 4.13.
Finalmente, en el último bucle `for`, actualizamos los pesos de salida, añadiendo además la influencia de los cambios que hubiese habido en la etapa anterior, a través del momento.

Para la capa oculta el procedimiento es exactamente el mismo, con la diferencia de que ahora nuestra delta depende de la delta anterior como indica la ecuación 4.23.
Tomando como si fuera nuestra entrada la salida que hemos recibido de la etapa de aprendizaje anterior, de nuevo volvemos a atravesar la función de activación al revés, como se puntualizó en la parte teórica, a través de su derivada.
De nuevo realizamos la multiplicación de los pesos, esta vez de entrada, por la entrada recibida de la última capa, y obtenemos una variable intermedia a la delta correspondiente a la capa oculta, que denominaremos `deltaHidden_aux`. Ésta vez tomaremos únicamente todas las filas excepto la primera, ya que, como recordamos, incluimos al principio una fila para los

umbrales que ya no será necesaria para el siguiente paso ya que será el de multiplicar por la matriz traspuesta de entrada del patrón para obtener nuestra delta de la capa oculta.
De nuevo, multiplicamos la delta por el ratio de aprendizaje para obtener la variación de nuestros pesos, y esta variación se la añadimos a los pesos que teníamos, además de la influencia de los cambios de la etapa anterior ponderada por el momento que se ha elegido.
Finalmente, se escriben los pesos tras haber sido modificados.

int main()

Es la función donde comienza el programa. Empezamos declarando variables tales como el tamaño de las matrices a utilizar y el patrón. También se inicializan los pesos de forma aleatoria y se elige la entrada y la salida deseada. Esto se hace para poder utilizar el módulo del perceptrón como un módulo aparte si se quiere utilizar con distintas entradas o salidas, o con más neuronas en la capa oculta.
Aunque los pesos estén inicializados de forma que siempre son los mismos, estos han sido generados a través de números aleatorios por el programa Matlab. Esto es debido a que si se inicializaban de la forma en la que está en los comentarios siempre se inicializaban los mismos pesos ya que la semilla era siempre la misma. Además, para poder comparar el éxito del programa con Matlab, se ha decidido utilizar semejantes pesos, pudiendo comprobar que ambos programas convergían en la misma iteración. (Elaboración propia a partir de CUDA Toolkit 2015, CPLUSPLUS 2015, Costa Fernandes 2015, Neural Networks and Deep Learning 2015, Isasi Viñuela, Galván León 2004 y Martínez Zarzuela 2015)

11. Presupuesto

Para la realización de este proyecto se ha utilizado:

Objeto	**Precio**
Procesador Intel Core i5-4690 CPU @ 3.50 GHz x 4	213,00 euros
Tarjeta gráfica Tesla C2075	990,00 euros
Procesador Intel® Xeon CPU X5650 @2.67GHz x 23	66,00 euros
3 Tarjetas gráficasGeForceGTX 580	3 x 471 euros
150 Horas sueldo ingeniero[1]	150 x 20 euros
Precio final	**5682,00 euros**

[1] Sueldo medio ingeniero según Pérez de Pablos 2015

12.Conclusiones y líneas futuras

En la investigación de las redes neuronales aún queda mucho por resolver. Si conseguimos adaptar las nuevas tecnologías de procesamiento en GPU a nuestras redes, éstas serán capaces de aprender mucho más rápido, y una de las barreras que hubo en los años 60 y por el cual muchos investigadores dejaron de lado las redes neuronales habrá desaparecido. Las redes neuronales artificiales pueden suponer a día de hoy una de las principales líneas de investigación en lo que a comprensión y creación del cerebro se refiere.

Se ha conseguido realizar el objetivo de este trabajo, pues tras haber realizado este proyecto, queda claro que tanto en el lenguaje CUDA como con la librería cuBLAS el rendimiento obtenido es mucho mayor. En uno de decenas, y en el otro llegando al millar.

Siguiendo la línea de esta investigación, el siguiente paso sería el de testear el perceptrón creado en un ejemplo real con pesos, entradas y salidas reales. Posteriormente, realizar un análisis del rendimiento en comparación con el rendimiento en GPU. Otra línea de investigación posible es la de una mejora del perceptrón multicapa híbrido, de forma que prácticamente todo fuera hecho en CUDA, es decir, no sólo la multiplicación y transposición de matrices, sino también las actualizaciones de pesos, las funciones de activación y la obtención de la regla delta (aunque esta en parte sea una multiplicación de matrices). Posteriormente, se debería analizar el perceptrón para datos reales, y finalmente comparar el rendimiento con CPU y con el perceptrón híbrido realizado en este proyecto.

13. Referencias bibliográficas

SÁNCHEZ CAMPEROS, E.N. y ALANÍS GARCÍA, A.Y., 2006. Redes neuronales. Conceptos fundamentales y aplicaciones a control automático. Pearson. Prentice Hall.

ISASI VIÑUELA, P. y GALVÁN LEÓN, I.M., 2004. Redes de Neuronas Artificiales. Un enfoque práctico. Pearson. Prentice Hall.

MARTÍN DEL BRÍO, B. y SANZ MOLINA, A., 2006. Redes neuronales y sistemas borrosos. Ra–Ma.

FLÓREZ LÓPEZ, R. y FERNÁNDEZ FERNÁNDEZ, J.M., 2008. Las redes neuronales artificiales. Fundamentos teóricos y aplicaciones prácticas.

NEURAL NETWORKS AND DEEP LEARNING (2015) Design of neural networks, how backpropagation algorithwoks and deep learning. Disponible en: http://neuralnetworksanddeeplearning.com/chap1.html (Consultado el 18 de Agosto de 2015)

COSTA FERNANDES, M.A., 2015. The matrix implementation of the Backpropagation algorithm for two-layer Multilayer Perceptron (MLP) neural networks, Universidade federal do Rio Grande do Norte. Departamento de ingeniería de computación y automoción.

KIRK, D. y HWU, W., 2010. Introduction to CUDA. Programming massively parallel processors. A hands-on approach. Morgan Kaufmann. Elsevier.

SANDERS, J. y KANDROT, E., 2011. CUDA by example. An introduction tu general purpose-GPu programming. Addison-Wesley.

CHRZĘSZCZYK, A. y CHRZĘSZCZYK, J., 2013. Matrix multiplications on the GPU. CUBLAS and MAGMA by example.

MARTÍNEZ ZARZUELA, M., 2015. Computación de propósito general en GPU, Universidad de Valladolid. Grupo de Telemática e Imagen GTI.

CUDA TOOLKIT (2015). CUDA Toolkit documentation. Disponible en: http://docs.nvidia.com/cuda/ (Consultado el 29 de agosto de 2015)

CUBLAS TOOLKIT (2015). cuBLAS Toolkit documentation. Disponible en: http://docs.nvidia.com/cuda/cublas/ (Consultado el 20 de julio de 2015)

NVIDIA DEVELOPER (2015). NVIDIA developer zone. All you need to develop with NVIDIA products. Disponible en: https://developer.nvidia.com/ (Consultado el 28 de agosto del 2015)

NVIDIA ESPAÑA (2015). Página de NVIDIA en España. Disponible en: http://www.nvidia.es/page/home.html (Consultado el 15 de mayo del 2015)

CPLUSPLUS (2015) General information about the C++ programming language. Disponible en: http://www.cplusplus.com/ (Consultado el 17 de abril del 2015)

ECLIPSE (2015) General information about the Eclipse interface. Disponible en: https://eclipse.org/ (Consultado el 17 de abril del 2015)

HELP ECLIPSE LUNA (2015) Eclipse help of the Eclipse Luna interface. Disponible en: http://help.eclipse.org/luna/index.jsp (Consultado el 17 de abril del 2015)

LIMONCELLO DIGITAL (2015) El mnemonista mecánico. Disponible en: http://www.limoncellodigital.com/2013_09_01_archive.html (Consultado el 2 de septiembre del 2015)

ABORATORIO INFORMÁTICA UC3M (2015) Características del perceptrón simple. Disponible en: http://www.lab.inf.uc3m.es/~a0080630/redes-de-neuronas/perceptron-simple.html (Consultado el 2 de septiembre del 2015)

CENSOR CÓSMICO (2015) Una forma sencilla de aprender Java, observando y deduciendo cómo se comporta el lenguaje a través de ejemplos prácticos. Disponible en: http://censorcosmico.blogspot.com.es/2014_02_01_archive.html (Consultado el 2 de septiembre del 2015)

PÉREZ DE PABLOS, S. 2012 "Se trabaja en equipo con gente que sabe muy bien lo que hace" Sociedad. El País. Disponible en: http://sociedad.elpais.com/sociedad/2012/08/28/actualidad/1346178826_082815.html (Consultado el 2 de septiembre de 2015)

14. Apéndices

14.1. Códigos en C++

14.1.1. Suma de vectores

```
#include <stdio.h>
```

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <iostream>

#define N 10
#define CUDA_CALL(x) do{\
	cudaError_t err = (x);\
	if(err!=cudaSuccess){\
		printf("Error \"%s\"\n",cudaGetErrorString(err));\
		exit(-1);\
	}}while(0)

__global__ void vectorAdd(int *A,int *B, int *C)
{
	//int tidx = blockIdx.x;
	int tidx = threadIdx.x + blockIdx.x*blockDim.x;
	C[tidx] = A[tidx] + B[tidx];
}
int main(void){
	size_t size = N*sizeof(int);

	int *dev_A,*dev_B,*dev_C=NULL;
	int A[N],B[N],C[N];
	cudaSetDevice(1);
	CUDA_CALL(cudaMalloc((void**)&dev_A,size));
	CUDA_CALL(cudaMalloc((void**)&dev_B,size));
	CUDA_CALL(cudaMalloc((void**)&dev_C,size));

	for(int i=0; i<N;i++){
		A[i] = -i;
		B[i] = i*i;
	}
	for(int j=0;j<N;j++){
		printf("%d ",A[j]);
	}
	printf("\n");
	for(int j=0;j<N;j++){
		printf("%d ",B[j]);
	}
	printf("\n");
	CUDA_CALL(cudaMemcpy(dev_A,A,size,cudaMemcpyHostToDevice));
	CUDA_CALL(cudaMemcpy(dev_B,B,size,cudaMemcpyHostToDevice));

	int blockdim = 256;
	int griddim = (N-1)/blockdim+1;
	printf("CUDA Kernel Launch with %d blocks and %d threads \n",
griddim, blockdim);
	vectorAdd<<<blockdim,griddim>>>(dev_A,dev_B,dev_C);
	cudaError_t err = cudaGetLastError();
	if(err!= cudaSuccess){
		printf("Failed to launch vector Add kernel");
		exit(EXIT_FAILURE);
	}
```

```
	CUDA_CALL(cudaMemcpy(C,dev_C,size,cudaMemcpyDeviceToHost));
	for(int j=0;j<N;j++){
		printf("%d ",C[j]);
	}
	CUDA_CALL(cudaFree(dev_A));
	CUDA_CALL(cudaFree(dev_B));
	CUDA_CALL(cudaFree(dev_C));
	//free(A);
	//free(B);
	//free(C);

	CUDA_CALL(cudaDeviceReset());
	return 0;
}
```

14.1.2. Multiplicación de matrices en CUDA

```
#include <stdio.h>
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <iostream>
#define IDX2C(i,j,ld) (((j)*(ld))+(i))

//Defining rows and cols values for each matrix

//Cuda macro for seeking errors
#define CUDA_CALL(x) do{\
	cudaError_t err = (x);\
	if(err!=cudaSuccess){\
		printf("Error \"%s\"\n",cudaGetErrorString(err));\
		exit(-1);\
	}}while(0)

//Code ran in the device
template <int BLOCK_SIZE> __global__ void
matrixMulCUDA(float *C, float *A, float *B, int wA, int wB)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;
    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int aBegin = wA * BLOCK_SIZE * by;
    int aEnd   = aBegin + wA - 1;
    int aStep  = BLOCK_SIZE;
    int bBegin = BLOCK_SIZE * bx;
    int bStep  = BLOCK_SIZE * wB;
    float Csub = 0;
    for (int a = aBegin, b = bBegin;
         a <= aEnd;
         a += aStep, b += bStep)
```

```
    {
        // Declaration of the shared memory array As
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        // Declaration of the shared memory array Bs
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];
        // Synchronize to make sure the matrices are loaded
        __syncthreads();
        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
#pragma unroll

        for (int k = 0; k < BLOCK_SIZE; ++k)
        {
            Csub += As[ty][k] * Bs[k][tx];
        }

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write the block sub-matrix to device memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
void matrixMulCPU(float *C, const float *A, const float *B, unsigned
int hA, unsigned int wA, unsigned int wB)
{
    for (unsigned int i = 0; i < hA; ++i)
        for (unsigned int j = 0; j < wB; ++j)
        {
            double sum = 0;

            for (unsigned int k = 0; k <wA; ++k)
            {
                double a = A[i * wA + k];
                double b = B[k * wB + j];
                sum += a * b;
            }

            C[i * wB + j] = (float)sum;
        }
}
bool compareMatrix(float *v1, float *v2, int sizeC, float tol){
      for(int i=0; i<sizeC; i++)
            if(v1[i] - v2[i] > tol)
                  return false;
```

```
      return true;
}
//Code ran in the CPU
int main(void){

      int nIter = 2;
      int average = 3;

      for(int loop = 0; loop < nIter; loop++){
            int iSize = 5 * (loop+1);
            printf("Simulation number %d\n\n",loop+1);

      int block=32;
      int hA = 2;
      int wA = 3;
      int hB = 3;
      int wB = 4;

      dim3 dimA(wA,hA,1);
      dim3 dimB(wB,hB,1);
    dim3 dimC(wB, hA, 1);

    printf("Matrixmult with CUDA..\n\n");
    printf("MatrixA(%u,%u), MatrixB(%u,%u), MatrixC(%u,%u)\n",
           dimA.x, dimA.y,
           dimB.x, dimB.y,
           dimC.x, dimC.y);

    if (dimA.x != dimB.y)
    {
        printf("Error: outer matrix dimensions must be equal. (%d !=
%d)\n",
               dimA.x, dimB.y);
        exit(EXIT_FAILURE);
    }
      //size declaration
      size_t sizeA = dimA.x*dimA.y*sizeof(float);
      size_t sizeB = dimB.x*dimB.y*sizeof(float);
      size_t sizeC = dimC.x*dimC.y*sizeof(float);
      //Variable declared
      float *d_A,*d_B,*d_C;
      //Allocate host memory
    float *h_A = (float *)malloc(sizeA);
    float *h_B = (float *)malloc(sizeB);
    float *h_C = (float *)malloc(sizeC);
    //Initializing host memory
    /*
    int j=0;
    for (j = 0; j < dimA.x*dimA.y; j++) {
      h_A[j] = rand()%2;
    }
    for (j = 0; j < dimB.x*dimB.y; j++) {
      h_B[j] = rand()%2;
```

```
}*/
h_B[0] = -1;
h_B[1] = -1;
h_B[2] = -1;
h_B[3] = -1;
h_B[4] = 0;
h_B[5] = 0;
h_B[6] = 1;
h_B[7] = 1;
h_B[8] = 0;
h_B[9] = 1;
h_B[10] = 0;
h_B[11] = 1;

h_A[0] = 0.8;
h_A[1] = 0.7;
h_A[2] = 0.2;
h_A[3] = 0.4;
h_A[4] = 0.2;
h_A[5] = 0.8;

//Printing parameters

printf("Matrix A:\n");
for(int i=0; i<dimA.y;i++){
  for(int j=0; j<dimA.x;j++){
        printf("%f ",h_A[i*dimA.x + j]);
  }
  printf("\n");
}
printf("Matrix B: \n");
for(int i=0; i<dimB.y;i++){
  for(int j=0; j<dimB.x;j++){
        printf("%f ", h_B[i*dimB.x + j]);
  }
  printf("\n");
}


//Creating event
cudaEvent_t startGPU, startCPU,stopGPU,stopCPU;
cudaEventCreate(&startGPU);
cudaEventCreate(&startCPU);
cudaEventCreate(&stopGPU);
cudaEventCreate(&stopCPU);

//Starting GPU record
cudaEventRecord(startGPU,0);


  //Memory allocation in the device
  cudaSetDevice(0);
  CUDA_CALL(cudaMalloc((void**)&d_A,sizeA));
```

```
  CUDA_CALL(cudaMalloc((void**)&d_B,sizeB));
  CUDA_CALL(cudaMalloc((void**)&d_C,sizeC));

  //Copy data from host to device
  CUDA_CALL(cudaMemcpy(d_A,h_A,sizeA,cudaMemcpyHostToDevice));
  CUDA_CALL(cudaMemcpy(d_B,h_B,sizeB,cudaMemcpyHostToDevice));

  //Preparing Threads and blocks
dim3 threads(block, block);
dim3 grid((wB-1) / threads.x+1, (hA-1) / threads.y+1);
  printf("CUDA Kernel Launched with success \n\n");

  for(int k=0; k <average;k++)
  {
  //Function to the device
  if(block==32){

  matrixMulCUDA<32><<<grid,threads>>>(d_C,d_A,d_B,dimA.x,dimB.x);
  }
  else{
        printf("Error with block size.\n");
  }
  cudaError_t err = cudaGetLastError();
  if(err!= cudaSuccess){
        printf("Failed to launch Matrix Mult kernel");
        exit(EXIT_FAILURE);
  }
  }
  CUDA_CALL(cudaMemcpy(h_C,d_C,sizeC,cudaMemcpyDeviceToHost));

  //Finishing GPU record
  cudaEventRecord(stopGPU,0);
  cudaEventSynchronize(stopGPU);
  float elapsedTimeGPU;
  cudaEventElapsedTime(&elapsedTimeGPU,startGPU,stopGPU);

  printf("Matrix C multiplied in GPU:\n");
  for(int i=0; i<dimB.x; i++){
        for(int j=0;j<dimA.y;j++){
              printf("%f ",h_C[i*dimB.x + j]);
        }
        printf("\n");
  }


  //Matrixmult in CPU

//Starting CPU record
cudaEventRecord(startCPU,0);
float *reference = (float *)malloc(sizeC);
matrixMulCPU(reference, h_A, h_B, hA, wA, wB);

  //Finishing GPU record
```

```
	cudaEventRecord(stopCPU,0);
	cudaEventSynchronize(stopCPU);
	float elapsedTimeCPU;
	cudaEventElapsedTime(&elapsedTimeCPU,startCPU,stopCPU);


	printf("Matrix C multiplied in CPU:\n");
	for(int i=0; i<dimB.x; i++){
		for(int j=0;j<dimA.y;j++){
			printf("%f ",reference[i*dimB.x + j]);
		}
		printf("\n");
	}

	// check result
	bool result =
compareMatrix(reference,h_C,sizeC/sizeof(float),1.0e-4f);

	printf("Comparing Matrix Multiply in GPU with CPU results:
%s\n", (true == result) ? "PASS" : "FAIL");
	// Compute and print the performance
	printf("GPU matrix multiplication:\n");
	float msecPerMatrixMul = elapsedTimeGPU / average;
	double flopsPerMatrixMul = 2.0 * (double)dimA.x * (double)dimA.y
* (double)dimB.x;
	double gigaFlops = (flopsPerMatrixMul * 1.0e-9f) /
(msecPerMatrixMul / 1000.0f);
	printf(
		"Performance= %.2f GFlop/s, Time= %.3f msec, Size= %.0f\n",
		gigaFlops,
		msecPerMatrixMul,
		flopsPerMatrixMul);
	double gigaFlopsCPU = (flopsPerMatrixMul * 1.0e-9f) /
(elapsedTimeCPU / 1000.0f);

	printf("CPU matrix multiplication:\n");
	printf(
		"Performance= %.2f GFlop/s, Time= %.3f msec, Size= %.0f\n",
		gigaFlopsCPU,
		elapsedTimeCPU,
		flopsPerMatrixMul);
	printf("Efficiency (GPU/CPU): %f\n", gigaFlops/gigaFlopsCPU);
	printf("CUDA Matrixmult sucessfully done.\n\n");
	cudaEventDestroy(startGPU);
	cudaEventDestroy(startCPU);
	cudaEventDestroy(stopGPU);
	cudaEventDestroy(stopCPU);

	CUDA_CALL(cudaFree(d_A));
	CUDA_CALL(cudaFree(d_B));
	CUDA_CALL(cudaFree(d_C));
	free(h_A);
	free(h_B);
```

```
	free(h_C);
	}
	return 0;

}
```

14.1.3. Multiplicación de matrices cuBLAS

```
#include <assert.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <stdio.h>
#include "device_launch_parameters.h"
#include <iostream>
#include <stdlib.h>

#define IDX2C(i,j,ld) (((j)*(ld))+(i))
//Cuda macro for seeking errors
#define CUDA_CALL(x) do{\
	cudaError_t err = (x);\
	if(err!=cudaSuccess){\
		printf("Error \"%s\"\n",cudaGetErrorString(err));\
		exit(-1);\
	}}while(0)


typedef struct _matrixSize      // Optional Command-line multiplier
for matrix sizes
{
    unsigned int uiWA, uiHA, uiWB, uiHB, uiWC, uiHC;
} sMatrixSize;
void
matrixMulCPU(float *C, const float *A, const float *B, unsigned int
hA, unsigned int wA, unsigned int wB)
{
    for (unsigned int i = 0; i < hA; ++i)
        for (unsigned int j = 0; j < wB; ++j)
        {
            double sum = 0;

            for (unsigned int k = 0; k <wA; ++k)
            {
                double a = A[k * hA + i];
                double b = B[j * wA + k];
                sum += a * b;
            }

            C[j * wB + i] = (float)sum;
        }
}
bool compareMatrix(float *v1, float *v2, int sizeC, float tol){
	for(int i=0; i<sizeC; i++)
		if(v1[i] - v2[i] > tol) return false;
	return true;
```

```
}
int main(int argc,char **argv){

      int nIter = 10;
      int average = 30;

      for(int loop = 0; loop < nIter; loop++){
            int iSize = 5 * (loop+1);
            printf("Simulation number %d\n\n",loop+1);
      //Declaring parameters
      printf("Initializing cuBLAS matrixmult...\n\n");

      int block_size = 32;
      sMatrixSize matrix_size;
    matrix_size.uiWA = 5 * block_size * iSize;
    matrix_size.uiHA = 3 * block_size * iSize;
    matrix_size.uiWB = 2 * block_size * iSize;
    matrix_size.uiHB = 5 * block_size * iSize;
    matrix_size.uiWC = 2 * block_size * iSize;
    matrix_size.uiHC = 3 * block_size * iSize;

    printf("MatrixA(%u,%u), MatrixB(%u,%u), MatrixC(%u,%u)\n\n",
           matrix_size.uiHA, matrix_size.uiWA,
           matrix_size.uiHB, matrix_size.uiWB,
           matrix_size.uiHC, matrix_size.uiWC);
    /*
      // Sizes must fit
      if(matrix_size.uiWA != matrix_size.uiHB){
            printf("Error: Matrix dimension mismatch. Width A: %d .
Height B : %d", matrix_size.uiWA,matrix_size.uiHB);
            return-1;
      }*/
      //Allocating host memory
      size_t sizeA = matrix_size.uiHA*matrix_size.uiWA*sizeof(float);
      size_t sizeB = matrix_size.uiHB*matrix_size.uiWB*sizeof(float);
      size_t sizeC = matrix_size.uiHC*matrix_size.uiWC*sizeof(float);

      float *d_A,*d_B,*d_C;
    float *h_A = (float *)malloc(sizeA);
    float *h_B = (float *)malloc(sizeB);
    float *h_C = (float *)malloc(sizeC);

    //Initializing parameters
    int j=0;
    for (j = 0; j < matrix_size.uiWA*matrix_size.uiHA; j++) {
      h_A[j] = rand()%2;
    }
    for (j = 0; j < matrix_size.uiWB*matrix_size.uiHB; j++) {
      h_B[j] = rand()%2;
    }

    // Printing parameters
    /*
```

```
    printf("Matrix A:\n");
    for(int i=0; i<matrix_size.uiHA;i++){
      for(int j=0; j<matrix_size.uiWA;j++){
            printf("%f ",h_A[j*matrix_size.uiHA + i]);
      }
      printf("\n");
    }
    printf("Matrix B: \n");
    for(int i=0; i<matrix_size.uiHB;i++){
      for(int j=0; j<matrix_size.uiWB;j++){
            printf("%f ", h_B[j*matrix_size.uiHB + i]);
      }
      printf("\n");
    }
    }*/

    //Creating event
    cudaEvent_t startGPU, startCPU,stopGPU,stopCPU;
    cudaEventCreate(&startGPU);
    cudaEventCreate(&startCPU);
    cudaEventCreate(&stopGPU);
    cudaEventCreate(&stopCPU);

    //Starting GPU record
    cudaEventRecord(startGPU,0);

      cudaSetDevice(0);
      CUDA_CALL(cudaMalloc((void**)&d_A,matrix_size.uiHA*matrix_size.u
iWA*sizeof(float)));
      CUDA_CALL(cudaMalloc((void**)&d_B,matrix_size.uiHB*matrix_size.u
iWB*sizeof(float)));
      CUDA_CALL(cudaMalloc((void**)&d_C,matrix_size.uiHC*matrix_size.u
iWC*sizeof(float)));

      cublasStatus_t stat;
      cublasHandle_t handle;
      stat = cublasCreate(&handle);

      stat =
cublasSetMatrix(matrix_size.uiHA,matrix_size.uiWA,sizeof(float),h_A,ma
trix_size.uiHA,d_A,matrix_size.uiHA);
      stat =
cublasSetMatrix(matrix_size.uiHB,matrix_size.uiWB,sizeof(float),h_B,ma
trix_size.uiHB,d_B,matrix_size.uiHB);
      stat =
cublasSetMatrix(matrix_size.uiHC,matrix_size.uiWC,sizeof(float),h_C,ma
trix_size.uiHC,d_C,matrix_size.uiHC);
      //CUDA_CALL(cudaMemcpy(d_A,h_A,sizeA,cudaMemcpyHostToDevice));
      //CUDA_CALL(cudaMemcpy(d_B,h_B,sizeB,cudaMemcpyHostToDevice));
      //CUDA_CALL(cudaMemcpy(d_C,h_C,sizeC,cudaMemcpyHostToDevice));
      const float alfa = 1.0f;
      const float beta = 0.0f;
    // setup execution parameters
```

```
    dim3 threads(block_size, block_size);
    dim3 grid((matrix_size.uiWC-1) / threads.x+1, (matrix_size.uiHC-1)
/ threads.y+1);
      for(int k=0; k <average;k++)
      {
    stat =
cublasSgemm(handle,CUBLAS_OP_N,CUBLAS_OP_N,matrix_size.uiHA,matrix_siz
e.uiWB,matrix_size.uiWA,&alfa,d_A,matrix_size.uiHA,d_B,matrix_size.uiH
B,&beta,d_C,matrix_size.uiHC);
      }
      stat =
cublasGetMatrix(matrix_size.uiHC,matrix_size.uiWC,sizeof(float),d_C,ma
trix_size.uiHC,h_C,matrix_size.uiHC);
      //CUDA_CALL(cudaMemcpy(h_C,d_C,sizeC,cudaMemcpyDeviceToHost));


      //Finishing GPU record
      cudaEventRecord(stopGPU,0);
      cudaEventSynchronize(stopGPU);
      float elapsedTimeGPU;
      cudaEventElapsedTime(&elapsedTimeGPU,startGPU,stopGPU);


      //Matrixmult in CPU
    cudaEventRecord(startCPU,0);
      float *reference = (float *)malloc(sizeC);
    matrixMulCPU(reference, h_A, h_B, matrix_size.uiHA,
matrix_size.uiWA, matrix_size.uiWB);

      //Finishing GPU record
      cudaEventRecord(stopCPU,0);
      cudaEventSynchronize(stopCPU);
      float elapsedTimeCPU;
      cudaEventElapsedTime(&elapsedTimeCPU,startCPU,stopCPU);

      //Printing C matrix
      /*
    printf("Matrix multiplied in GPU:\n\n");
    for(int i=0; i<matrix_size.uiHC;i++){
      for(int j=0; j<matrix_size.uiWC;j++){
            printf("%f ", h_C[j*matrix_size.uiHC + i]);
      }
      printf("\n");
    }

    printf("Matrix multiplied in CPU:\n\n");
    for(int i=0; i<matrix_size.uiHC;i++){
      for(int j=0; j<matrix_size.uiWC;j++){
            printf("%f ", reference[j*matrix_size.uiHC + i]);
      }
      printf("\n");
    }
    }*/
```

```
      // check result
            bool result =
compareMatrix(reference,h_C,sizeC/sizeof(float),1.0e-4f);

            printf("Comparing Matrix Multiply in GPU with CPU results:
%s\n", (true == result) ? "PASS" : "FAIL");
            // Compute and print the performance
            printf("GPU matrix multiplication:\n");
            float msecPerMatrixMul = elapsedTimeGPU / average;
            double flopsPerMatrixMul = 2.0 * (double)matrix_size.uiWA *
(double)matrix_size.uiHA * (double)matrix_size.uiWB;
            double gigaFlops = (flopsPerMatrixMul * 1.0e-9f) /
(msecPerMatrixMul / 1000.0f);
            printf(
                  "Performance= %.2f GFlop/s, Time= %.3f msec, Size=
%.0f\n",
                  gigaFlops,
                  msecPerMatrixMul,
                  flopsPerMatrixMul);
            double gigaFlopsCPU = (flopsPerMatrixMul * 1.0e-9f) /
(elapsedTimeCPU / 1000.0f);

            printf("CPU matrix multiplication:\n");
            printf(
                  "Performance= %.2f GFlop/s, Time= %.3f msec, Size=
%.0f\n",
                  gigaFlopsCPU,
                  elapsedTimeCPU,
                  flopsPerMatrixMul);
            printf("Efficiency (GPU/CPU): %f\n\n",
gigaFlops/gigaFlopsCPU);

      cudaEventDestroy(startGPU);
      cudaEventDestroy(startCPU);
      cudaEventDestroy(stopGPU);
      cudaEventDestroy(stopCPU);

    CUDA_CALL(cudaFree(d_A));
    CUDA_CALL(cudaFree(d_B));
    CUDA_CALL(cudaFree(d_C));
    cublasDestroy(handle);
    free(h_A);
    free(h_B);
    free(h_C);

    printf("CuBLAS matrixmult successfully done.\n\n");
      }
    return 0;

}
```

14.1.4. Resolución de sistemas lineales

```
#include <assert.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <stdio.h>
#include "device_launch_parameters.h"
#include <iostream>
#include <stdlib.h>

#define IDX2C(i,j,ld) (((j)*(ld))+(i))
//Cuda macro for seeking errors
#define CUDA_CALL(x) do{\
      cudaError_t err = (x);\
      if(err!=cudaSuccess){\
            printf("Error \"%s\"\n",cudaGetErrorString(err));\
            exit(-1);\
      }}while(0)

// Solving a Sistem Ax = b using cuBLAS library
// Matrix A has n x n dimensions
// Vector x has n x 1 dimensions
// Vector y has n x 1 dimensions


int main(int argc,char **argv){

      cudaError_t cudaStat;
      cublasStatus_t stat ;
      cublasHandle_t handle ;
      int n = 30;
      float* a;
      float* x;
      float *y;

      a = (float *)malloc(n*n*sizeof(float));
      x = (float *)malloc(n*sizeof(float));
      y = (float *)malloc(n*sizeof(float));

      float *d_a,*d_x;

      int ind = 1;
      for(int i=0; i<n;i++){
            for(int j=0;j<n;j++){
                  if(i>=j)
                        a[IDX2C(i,j,n)] = ind++;
                  else
                        a[IDX2C(i,j,n)] = 0;
            }
      }
      for (int i=0; i<n ; i++) x [i]=1.0f;

    printf("Matrix A:\n");
    for(int i=0; i<n;i++){
```

```
      for(int j=0; j<n;j++){
            printf("%f ",a[j*n + i]);
      }
      printf("\n");
    }
      CUDA_CALL(cudaMalloc((void**)&d_a,n*n*sizeof(float)));
      CUDA_CALL(cudaMalloc((void**)&d_x,n*sizeof(float)));

      stat = cublasCreate(&handle);
      stat = cublasSetMatrix (n,n,sizeof(float),a,n,d_a,n);
      stat = cublasSetVector (n,sizeof(float),x,1,d_x,1);

      stat=cublasStrsv(handle,CUBLAS_FILL_MODE_LOWER,CUBLAS_OP_N,CUBLA
S_DIAG_NON_UNIT,n,d_a,n,d_x,1);
      stat = cublasGetVector(n,sizeof(float),d_x,1,y,1);

      printf ("Vector solution: \n");
      for ( int j =0; j < n ; j ++){
            printf ("%f ",y[j]);
            printf ("\n");
      }
      cudaFree(d_a);
      cudaFree(d_x);
      cublasDestroy(handle);
      free(a);
      free(x);
      free(y);
      return EXIT_SUCCESS ;
}
```

14.1.5. Factorización LU en cuBLAS

```
#include <assert.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <stdio.h>
#include "device_launch_parameters.h"
#include <iostream>
#include <stdlib.h>

#define IDX2C(i,j,ld) (((j)*(ld))+(i))
//Cuda macro for seeking errors
#define CUDA_CALL(x) do{\
      cudaError_t err = (x);\
      if(err!=cudaSuccess){\
            printf("Error \"%s\"\n",cudaGetErrorString(err));\
            exit(-1);\
      }}while(0)

// Solving a Sistem A*x = b using cuBLAS library and LU factorization
// We already have LU factorization and ! = L*U
// Matrix L has n x n dimensions upper diag
```

```
// MAtrix U has n x n dimensions lower diag
// We solve 1) U*x = y and 2) L*y = b
// Vector x has n x 1 dimensions
// Vector b has n x 1 dimensions

int main(int argc,char **argv){

      cudaError_t cudaStat;
      cublasStatus_t stat ;
      cublasHandle_t handle ;
      int n = 3;
      float* L;
      float* U;
      float* x;
      float *b;

      L = (float *)malloc(n*n*sizeof(float));
      U = (float *)malloc(n*n*sizeof(float));
      x = (float *)malloc(n*sizeof(float));
      b = (float *)malloc(n*sizeof(float));

      float *d_L,*d_U,*d_x;

      int ind = 1;
      for(int i=0; i<n;i++){
            for(int j=0;j<n;j++){
                  if(i<=j)
                        L[IDX2C(i,j,n)] = ind++;
                  else
                        L[IDX2C(i,j,n)] = 0;
            }
      }
      ind = 1;
      for(int i=0; i<n;i++){
            for(int j=0;j<n;j++){
                  if(i>=j)
                        U[IDX2C(i,j,n)] = ind++;
                  else
                        U[IDX2C(i,j,n)] = 0;
            }
      }
      for (int i=0; i<n ; i++) x [i]=1.0f;

    printf("Matrix L:\n");
    for(int i=0; i<n;i++){
      for(int j=0; j<n;j++){
            printf("%f ",L[j*n + i]);
      }
      printf("\n");
    }
    printf("Matrix U:\n");
    for(int i=0; i<n;i++){
      for(int j=0; j<n;j++){
```

```
            printf("%f ",U[j*n + i]);
        }
        printf("\n");
    }
        CUDA_CALL(cudaMalloc((void**)&d_L,n*n*sizeof(float)));
        CUDA_CALL(cudaMalloc((void**)&d_U,n*n*sizeof(float)));
        CUDA_CALL(cudaMalloc((void**)&d_x,n*sizeof(float)));

        stat = cublasCreate(&handle);
        stat = cublasSetMatrix (n,n,sizeof(float),L,n,d_L,n);
        stat = cublasSetMatrix (n,n,sizeof(float),U,n,d_U,n);
        stat = cublasSetVector (n,sizeof(float),x,1,d_x,1);

        stat=cublasStrsv(handle,CUBLAS_FILL_MODE_LOWER,CUBLAS_OP_N,CUBLA
S_DIAG_NON_UNIT,n,d_U,n,d_x,1);
        stat=cublasStrsv(handle,CUBLAS_FILL_MODE_UPPER,CUBLAS_OP_N,CUBLA
S_DIAG_NON_UNIT,n,d_L,n,d_x,1);
        stat = cublasGetVector(n,sizeof(float),d_x,1,b,1);

        printf ("Vector solution: \n");
        for ( int j =0; j < n ; j ++){
            printf ("%f ",b[j]);
            printf ("\n");
        }
        cudaFree(d_L);
        cudaFree(d_U);
        cudaFree(d_x);
        cublasDestroy(handle);
        free(L);
        free(U);
        free(x);
        free(b);
        return EXIT_SUCCESS ;
}
```

14.1.6. Inversión de matrices

```
#include <cstdio>
#include <cstdlib>
#include <cuda_runtime.h>
#include <cublas_v2.h>

#define cudacall(call)
    do
    {
        cudaError_t err = (call);
        if(cudaSuccess != err)
        {
            fprintf(stderr,"CUDA Error:\nFile = %s\nLine = %d\nReason
= %s\n", __FILE__, __LINE__, cudaGetErrorString(err));
            cudaDeviceReset();
            exit(EXIT_FAILURE);
        }
    }
    while (0)
#define cublascall(call)
```

```
    do
    {
        cublasStatus_t status = (call);
        if(CUBLAS_STATUS_SUCCESS != status)
        {
            fprintf(stderr,"CUBLAS Error:\nFile = %s\nLine = %d\nCode
= %d\n", __FILE__, __LINE__, status);
            cudaDeviceReset();
            exit(EXIT_FAILURE);
        }
    }
    while(0)

void cublas_inv(int n, double* a)
{
    cublasHandle_t handle;
    cublascall(cublasCreate(&handle));
    double **PtrA = 0;
    double **PtrA_dev = NULL;
    int *d_pivot_array;
    int *d_info_array;
    int Batchsize = 1;
    double *y,*x;
    y = (double *)malloc(n*sizeof(double));
    x = (double *)malloc(n*sizeof(double));

    cudacall(cudaMalloc((void **)&d_pivot_array, n * sizeof(int)));
    cudacall(cudaMalloc((void **)&d_info_array, sizeof(int)));
    PtrA =(double **)malloc(Batchsize * sizeof(*PtrA));

    cudacall(cudaMalloc((void **) PtrA, n*n * sizeof(double)));
    cudacall(cudaMalloc((void **) &PtrA_dev, Batchsize *
sizeof(*PtrA)));

    cudacall(cudaMemcpy(PtrA_dev, PtrA, Batchsize * sizeof(*PtrA),
cudaMemcpyHostToDevice));

    cublascall(cublasSetMatrix(n, n, sizeof(a[0]), a, n, PtrA[0], n));

    cublascall(cublasDgetrfBatched(handle, n,
PtrA_dev,n,d_pivot_array,d_info_array,Batchsize));

    cublascall(cublasGetMatrix(n, n, sizeof(double), PtrA[0], n, a,
n));

      cublascall(cublasGetVector(n,sizeof(int),d_pivot_array,1,x,1));
      printf ("Pivot Array:\n ");
      for ( int j =0; j < n*Batchsize ; j ++){
            printf ("%f ",x[j]);
            printf ("\n");
      }

      cublascall(cublasGetVector(Batchsize,sizeof(int),d_info_array,1,
y,1));
      printf ("Info_Array:\n ");
```

```
      for ( int j =0; j < Batchsize ; j ++){
      printf ("%f ",y[j]);
      printf ("\n");
      }
}


int main()
{
      const int n = 4;
      double A[n * n] = { 1.0, 2.0, 2.0, 2.0,
                                    2.0, 1.0, 2.0, 2.0,
                                    2.0, 2.0, 1.0, 2.0,
                                    2.0, 2.0, 2.0, 1.0 };
      double* a = A;

      fprintf(stdout, "Input:\n\n");
    for(int i=0; i<n; i++)
    {
        for(int j=0; j<n; j++)
            fprintf(stdout,"%f\t",A[i*n+j]);
        fprintf(stdout,"\n");
    }

      cublas_inv(n, a);

      fprintf(stdout, "Output:\n\n");
    for(int i=0; i<n; i++)
    {
        for(int j=0; j<n; j++)
            fprintf(stdout,"%f\t",A[i*n+j]);
        fprintf(stdout,"\n");
    }
}
```

14.1.7. Perceptrón multicapa híbrido

```
#include "stdio.h"
#include <iostream>
#include <stdlib.h>
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include "math.h"
#define IDX2C(i,j,ld) (((j)*(ld))+(i))
#define TOL 0.001

//Defining rows and cols values for each matrix

//Cuda macro for seeking errors
#define CUDA_CALL(x) do{\
      cudaError_t err = (x);\
      if(err!=cudaSuccess){\
            printf("Error \"%s\"\n",cudaGetErrorString(err));\
            exit(-1);\
```

```
      }}while(0)

//Code ran in the device
template <int BLOCK_SIZE> __global__ void
matrixMulCUDA(float *c, float *a, float *b, int wA, int wB)
{
        int bx     = blockIdx.x,  by = blockIdx.y;
        int tx     = threadIdx.x, ty = threadIdx.y;
        int aBegin = wA * BLOCK_SIZE * by;
        int aEnd   = aBegin + wA - 1;
        int bBegin = BLOCK_SIZE * bx;
        int aStep  = BLOCK_SIZE, bStep  = BLOCK_SIZE * wB;
        float sum  = 0.0f;
        for ( int ia = aBegin, ib = bBegin; ia <= aEnd; ia += aStep,
ib += bStep )
        {
          __shared__ float as [BLOCK_SIZE][BLOCK_SIZE];
          __shared__ float bs [BLOCK_SIZE][BLOCK_SIZE];
          as [ty][tx] = a [ia + wA * ty + tx];
          bs [ty][tx] = b [ib + wA * ty + tx];
          __syncthreads ();   // Synchronize to make sure the matrices
are loaded
          for ( int k = 0; k < BLOCK_SIZE; k++ ) sum += as [ty][k] *
bs [k][tx];
          __syncthreads ();   // Synchronize to make sure submatrices
not needed
        }
        c [wB * BLOCK_SIZE * by + BLOCK_SIZE * bx + wB * ty + tx] =
sum;
}
//Code ran in the CPU
int matrixmult(float* h_C,float* h_B,float* h_A, int hA,int wA, int
wB){

      int block=1;
      int hB = wA;

      dim3 dimA(wA,hA,1);
      dim3 dimB(wB,hB,1);
    dim3 dimC(wB, hA, 1);

    if (dimA.x != dimB.y)
    {
        printf("Error: outer matrix dimensions must be equal. (%d !=
%d)\n",
               dimA.x, dimB.y);
        exit(EXIT_FAILURE);
    }
      //size declaration
      size_t sizeA = dimA.x*dimA.y*sizeof(float);
      size_t sizeB = dimB.x*dimB.y*sizeof(float);
      size_t sizeC = dimC.x*dimC.y*sizeof(float);
      //Variable declared
```

```
      float *d_A,*d_B,*d_C;

      //Memory allocation in the device
      cudaSetDevice(0);
      CUDA_CALL(cudaMalloc((void**)&d_A,sizeA));
      CUDA_CALL(cudaMalloc((void**)&d_B,sizeB));
      CUDA_CALL(cudaMalloc((void**)&d_C,sizeC));

      //Copy data from host to device
      CUDA_CALL(cudaMemcpy(d_A,h_A,sizeA,cudaMemcpyHostToDevice));
      CUDA_CALL(cudaMemcpy(d_B,h_B,sizeB,cudaMemcpyHostToDevice));

      //Preparing Threads and blocks
    dim3 threads(block, block);
    dim3 grid((wB-1) / threads.x+1, (hA-1) / threads.y+1);

      //Function to the device
      if(block==1){
      matrixMulCUDA<1><<<grid,threads>>>(d_C,d_A,d_B,dimA.x,dimB.x);
      }
      else{
            printf("Error with block size.\n");
      }
      cudaError_t err = cudaGetLastError();
      if(err!= cudaSuccess){
            printf("Failed to launch Matrix Mult kernel");
            exit(EXIT_FAILURE);
      }
      CUDA_CALL(cudaMemcpy(h_C,d_C,sizeC,cudaMemcpyDeviceToHost));
      CUDA_CALL(cudaFree(d_A));
      CUDA_CALL(cudaFree(d_B));
      CUDA_CALL(cudaFree(d_C));
      return 0;

}

template< int TILE_DIM>__global__ void transpose_dev(float *odata,
float *idata)
{
      const int BLOCK_ROWS=8;

  __shared__ float tile[TILE_DIM][TILE_DIM];

  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];

  __syncthreads();

  x = blockIdx.y * TILE_DIM + threadIdx.x;  // transpose block offset
```

```
  y = blockIdx.x * TILE_DIM + threadIdx.y;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}
void transpose(float *odata, const float *idata, int dim)
{
      float *idata_dev;
      float *odata_dev;
      int block=32;
      //Memory allocation in the device
      cudaSetDevice(0);
      CUDA_CALL(cudaMalloc((void**)&idata_dev,dim));
      CUDA_CALL(cudaMalloc((void**)&odata_dev,dim));

      //Copy data from host to device
      CUDA_CALL(cudaMemcpy(idata_dev,idata,dim,cudaMemcpyHostToDevice)
);
      CUDA_CALL(cudaMemcpy(odata_dev,odata,dim,cudaMemcpyHostToDevice)
);

      //Preparing Threads and blocks
    dim3 threads(block, block);
    dim3 grid((dim-1) / threads.x+1, (dim-1) / threads.y+1);
    //Function to the device
      if(block==32){
      transpose_dev<32><<<grid,threads>>>(odata_dev,idata_dev);
      }
      else{
            printf("Error with block size.\n");
      }
      cudaError_t err = cudaGetLastError();
      if(err!= cudaSuccess){
            printf("Failed to launch Matrix Mult kernel");
            exit(EXIT_FAILURE);
      }
      CUDA_CALL(cudaMemcpy(odata,odata_dev,dim,cudaMemcpyDeviceToHost)
);
      CUDA_CALL(cudaFree(idata_dev));
      CUDA_CALL(cudaFree(odata_dev));

}

/*
void transpon(float *data_transp,float *data, int rows, int cols){
      for (int i = 0; i < rows; i++)
              for (int j = 0; j < cols; j++)
                  data_transp[(j*rows)+i] = data[(i*cols)+j];
}*/
void perceptron(float *input,float *salida_deseada,float
momentum,float rate,int Niter, int n, int iter,float* weightsInput,
float* weightsOutput,float* weightsInput_ant,float* weightsOutput_ant,
int patron){
```

```
      int m = n+1;
      float *delta = (float *)malloc(m*sizeof(float));
      float *delta_aux = (float *)malloc(m*sizeof(float));
      float *deltaHidden = (float *)malloc(n*m*sizeof(float));
      float *deltaHidden_aux = (float
*)malloc(m*patron*sizeof(float));
      float *deltaHidden_aux_2 = (float
*)malloc(n*patron*sizeof(float));
      float *err = (float *)malloc(Niter*sizeof(int));
      float *InputsHidden = (float *)malloc(patron*n*sizeof(float));
      float *d_InputsHiddenTAN = (float
*)malloc(patron*m*sizeof(float));
      float *InputsHiddenTAN = (float
*)malloc(patron*m*sizeof(float));
      float *InputsHiddenTAN_transp = (float
*)malloc(patron*m*sizeof(float));
      float *OutputsHidden = (float *)malloc(patron*sizeof(float));
      float *d_OutputsHidden = (float *)malloc(patron*sizeof(float));
      float *newInput = (float *)malloc(m*patron*sizeof(float));
      float *newInput_transp = (float
*)malloc(m*patron*sizeof(float));
      float *newInput_aux = (float *)malloc(m*sizeof(float));
      float *weightsInput_aux = (float *)malloc(n*m*sizeof(float));
      float *weightsOutput_aux = (float *)malloc(m*sizeof(float));
      float *weightsOutput_transp = (float *)malloc(m*sizeof(float));

      float error[4];

      /*
    for (int i = patron; i > 0; i--) {
            for (int j = n; j > 0; j--) {
                  newInput[i*m-j-1] = input[i*n-j];
            }
                  newInput[i*(patron-1)-1] = -1;
    }*/
    for (int i = 0; i < n*patron; i++) {
      newInput[i+patron] = input[i];
    }
    for (int i = 0; i < patron; i++) {
      newInput[i] = -1;
    }
  //  for(int k=0;k<patron;k++){
      //newInput_aux[0] = newInput[m*k];
      //newInput_aux[1] = newInput[m*k+1];
      //    for(int k=0;k<patron;k++){newInput_aux[2] =
newInput[m*k+2];
            //Training
      matrixmult(InputsHidden,newInput,weightsInput,n,m,patron);
    //}

      for(int j=0;j<n*patron;j++){
            InputsHiddenTAN[j] = 1/(1+exp(-InputsHidden[j]));
```

```
	}

	for (int i = n*patron-1; i >= 0; i--) {
		InputsHiddenTAN[i+patron] = InputsHiddenTAN[i];
	}

	for (int i = 0; i < patron; i++) {
		InputsHiddenTAN[i] = -1;
	}


	matrixmult(OutputsHidden,InputsHiddenTAN,weightsOutput,1,m,patro
n);
	for(int j=0;j<patron;j++){
		OutputsHidden[j] = 1/(1+exp(-OutputsHidden[j]));
	}
	for(int k=0;k<patron;k++){
		error[k] = salida_deseada[k] - OutputsHidden[k];
	}
	printf("Weights before:\n InputWeigths: [w11] = %f, [w12] = %f,
[w21] = %f, [w22] = %f, Threshold: [u1] = %f, [u2] = %f, [u3] = %f,
OutputWeights: [v1] = %f, [v2] = %f\n",
weightsInput[0],weightsInput[1],weightsInput[3],weightsInput[4],weight
sInput[2],weightsInput[5],weightsOutput[2],weightsOutput[0],weightsOut
put[1]);


	// Result comparison
	double mse = 0;
	for(int k=0;k<patron;k++){
		mse = mse + pow(error[k],2);
	}
	mse = mse/patron;

	printf("Error MSE: %f\n", mse);
	if(mse < TOL){
		printf("Converged on iteration %d\n", iter);
		exit(0);
	}
	// Learning
	// Output Layer
	for(int k=0;k<patron;k++){
		d_OutputsHidden[k] =  OutputsHidden[k] * (1 -
OutputsHidden[k]);
		d_OutputsHidden[k] = d_OutputsHidden[k] * error[k];
	}
	transpose(InputsHiddenTAN_transp,InputsHiddenTAN, m* patron);

	matrixmult(delta_aux,InputsHiddenTAN_transp,d_OutputsHidden,1,pa
tron,m);
	for(int k=0;k<m;k++){
		delta[k] =  (rate/patron)*delta_aux[k];
	}
```

```
      for(int k=0;k<m;k++){
            weightsOutput_aux[k] = weightsOutput[k];
            weightsOutput[k] = weightsOutput[k] + delta[k] +
momentum*weightsOutput_ant[k];
            weightsOutput_ant[k] = weightsOutput_aux[k];
      }

      // Hidden Layer
      for(int k=0;k<m*patron;k++){
            d_InputsHiddenTAN[k] =  InputsHiddenTAN[k] * (1 -
InputsHiddenTAN[k]);
      }
      transpose(weightsOutput_transp,weightsOutput,  m);

      matrixmult(deltaHidden_aux,d_OutputsHidden,weightsOutput_transp,
m,1,patron);
      for(int k=0;k<m*patron;k++){
            deltaHidden_aux[k] =
d_InputsHiddenTAN[k]*deltaHidden_aux[k];
      }
      for(int k=patron;k<m*patron;k++){
            deltaHidden_aux_2[k-patron] =  deltaHidden_aux[k];
      }

      transpose(newInput_transp,newInput, m* patron);

      matrixmult(deltaHidden,newInput_transp,deltaHidden_aux_2,n,patro
n,m);
      for(int k=0;k<m*n;k++){
            deltaHidden[k] =  (rate/patron)*deltaHidden[k];
      }
      for (int k = 0; k < n*m; k++) {
            weightsInput_aux[k] = weightsInput[k];
            weightsInput[k] = weightsInput[k] + deltaHidden[k] +
momentum * weightsInput_ant[k];
            weightsInput_ant[k] = weightsInput_aux[k];
          }

          /*
            for(int l=0;l<6;l++){
            printf("El numero: %f\n",weightsInput[l]);
            }

            matrixmult(delta_aux,newInput,weightsInput,n,m,1);
            for(int j=0;j<n;j++){
                  delta_aux[j] = (tanh(delta_aux[j])+1)*(1-
(tanh(delta_aux[j]) + 1)/2);
                  deltaHidden[j] = delta_aux[j] * delta[0] *
weightsOutput[j];
            }
            matrixmult(deltaHidden_aux,newInput,deltaHidden,n,1,m);
            //Refreshing weights
            for(int j=0;j<m;j++){
```

```
            weightsOutput[j] = weightsOutput[j] + rate * delta[0] *
InputsHiddenTAN[j];
            }
          for (int i = 0; i < n; i++)
          {
                  for(int j=0;j<m;j++){
                        weightsInput[IDX2C(j,i,m)] =
weightsInput[IDX2C(j,i,m)] + rate * deltaHidden_aux[IDX2C(j,i,m)];
                  }
          }
            */
printf("Weights after:\n InputWeigths: [w11] = %f, [w12] = %f, [w21] =
%f, [w22] = %f, Threshold: [u1] = %f, [u2] = %f, [u3] = %f,
OutputWeights: [v1] = %f, [v2] = %f\n",
weightsInput[0],weightsInput[1],weightsInput[3],weightsInput[4],weight
sInput[2],weightsInput[5],weightsOutput[2],weightsOutput[0],weightsOut
put[1]);
}

int main()
{
      // Inicializacion de variables

      float rate = 0.75;
      float momentum = 0.001;
      int Niter = 5000;
      int longitud = 2;
      int patron = 4;
      float salida_deseada[4] = {0,1,1,0};
      float *weightsInput = (float
*)malloc(longitud*(longitud+1)*sizeof(float));
      float *weightsOutput = (float
*)malloc((longitud+1)*sizeof(float));
      float *input = (float *)malloc(patron*longitud*sizeof(float));
      float *weightsInput_ant = (float
*)malloc(longitud*(longitud+1)*sizeof(float));
      float *weightsOutput_ant = (float
*)malloc((longitud+1)*sizeof(float));

      for (int i = 0; i < longitud*(longitud+1); i++) {
            weightsInput_ant[i] = 0;
      }
      for (int i = 0; i < longitud+1; i++) {
            weightsOutput_ant[i] = 0;
      }

          input[0] = 0;
          input[1] = 0;
          input[2] = 1;
          input[3] = 1;
          input[4] = 0;
          input[5] = 1;
          input[6] = 0;
```

```
            input[7] = 1;
      //Generate random weights
    /*
    for (int i = 0; i < longitud; i++) {
            for (int j = 0; j < longitud+1; j++) {
                  weightsInput[i*(longitud+1)+j] = (1+rand()%8)*0.1f;
            }
    }
      for (int j = 0; j < longitud+1; j++) {
            weightsOutput[j] = (1+rand()%8)*0.1f;
      }*/
       weightsInput[0] = 0.1754;
      weightsInput[1] = 0.9768;
      weightsInput[2] = 0.2494;
      weightsInput[3] = 0.0741;
      weightsInput[4] = 0.2436;
      weightsInput[5] = 0.8305;
      weightsOutput[0] = 0.8518;
      weightsOutput[1] = 0.0355;
      weightsOutput[2] = 0.4480;


      printf("Initializing CUDA perceptron\n\n");
      for(int iter = 0; iter <Niter;iter++)
      {
            printf("Iteration Number %d\n",iter);
      perceptron(input,salida_deseada,momentum,rate,Niter,
longitud,iter,
weightsInput,weightsOutput,weightsInput_ant,weightsOutput_ant,patron);
      }
      return 0;
}
```

14.2. Ejercicios Realizados de CUDA

Se han realizado unos ejercicios sobre la multiplicación de matrices y CUDA en general, para aquellos alumnos que están iniciándose con la herramienta CUDA. A continuación se desarrollan tanto las preguntas como las respuestas de los ejercicios.

Ejercicios de búsqueda de fallos en el código:

Ejercicio 1:

Sea el código de la figura:

```
#include <iostream>
#include <stdio.h>
#include "cuda_runtime.h"


#define CUDA_CALL(x) do{\
    cudaError_t err = (x);\
    if(err!=cudaSuccess){\
        printf("Error \"%s\"\n",cudaGetErrorString(err));\
        exit(-1);\
    }}while(0)

__global__ void add(int a,int b, int *c){
    *c = a+b;
}

int main(void){
    int c;
    int *dev_c;
    add<<<1,1>>>(2,7,dev_c);

    CUDA_CALL(cudaMemcpy(&c,dev_c,sizeof(int),cudaMemcpyDeviceToHost));
    printf("2+7=%d\n",c);
    cudaFree(dev_c);
    return 0;
}
```

al compilarlo nos da el siguiente warning:

```
../hello_world.cu(21): warning: variable "dev_c" is used before its value is set
```

Al ejecutar el código, ¿podrá el programa funcionar correctamente? En caso de que si, explique por qué este warning no es imprescindible para que el código funcione. En caso de que no, diga qué habría que añadir para que el código funcionara correctamente.

Respuesta:

Al ejecutarse, no funcionará debido a que la variable de la dirección en el *device* no está inicializada. Para ello, se podría inicializar a mano, con la línea de código:

```
Int *dev_c = NULL;
```

Sin embargo, de esa forma nos faltaría reservar memoria en el *device* para poder hacer la cuenta correctamente. Por ello habría que añadir un código con la función `cudaMalloc()`.

Un ejemplo de ese código podría ser la siguiente línea:

```
CUDA_CALL (cudaMalloc((void **)&dev_c, sizeof(int)));
```

De esta forma reservamos el espacio, inicializamos y obtenemos por pantalla el resultado esperado:

2+7=9

Ejercicio 2:

¿Qué falla en el código de la figura para que no se pueda ejecutar correctamente?

```
int main(void){
    int c;
    int *dev_c;

    CUDA_CALL(cudaMalloc((void **)&dev_c,sizeof(int)));
    add<<<1,1>>>(2,7,dev_c);

    CUDA_CALL(cudaMemcpy(&c,dev_c,sizeof(int),cudaMemcpyHostToDevice));
    printf("2+7=%d\n",c);
    cudaFree(dev_c);
    return 0;
}
```

Suponga que la función `add()` está definida y funciona correctamente.

Respuesta:

Tras realizar correctamente la función `add()`, la respuesta se encuentra en la posición `dev_c` de la memoria del *device*. Sin embargo, al hacer un `cudaMemcpy()` estamos copiando desde el *host* al *device*, cuando lo que queremos es copiar del *device* al *host*. Habría que cambiar esa parte del código, de forma que la línea quedaría así:

```
CUDA_CALL(cudaMemcpy(&c,dev_c,sizeof(int),cudaMemcpyDeviceToHost
));
```

Ejercicios de cuestiones teóricas:

Ejercicio 1:

Se desea realizar una suma de matrices cuadradas A + B de forma óptima donde la dimensión de éstas es de 1024. Para ello, se ha decidido utilizar una dimensión de bloque de 64 *threads* (64x64). ¿De cuántos bloques dispondrá el *grid* que hemos decidido crear? ¿Cuántos *warps* habrá en cada bloque? ¿Y en todo el *grid*?

Respuesta:

Tenemos una dimensión de 1024x 1024 en ambas matrices. Eso quiere decir que necesitaremos un total de 1024/64= 16 bloques en la dirección x, y otros 16 bloques en la dirección y. Esto hace un total de 256 bloques.

Los *warps* son agrupaciones de 32 *threads*, como cada bloque tiene 64x64 *threads*, cada bloque tendrá un total de 4096 *threads*. Si los agrupamos en paquetes de 32, Obtenemos un total de 4096/32= 128 *warps*.

Tenemos un total de 256 bloques, cada bloque son 128 *warps*. En total tendremos 32.768 *warps* en el *grid*.

Ejercicio 2:

Explique detalladamente para qué sirve la siguiente macro de CUDA:

```
#define CUDA_CALL(x) do{\
    cudaError_t err = (x);\
    if(err!=cudaSuccess){\
        printf("Error \"%s\"\n",cudaGetErrorString(err));\
        exit(-1);\
    }}while(0)
```

Respuesta:

El código hace un *do…while* en el que se ejecuta siempre el código que está dentro. En su interior, declaramos una variable de tipo `cudaError_t`, que la llamamos `err` y que le damos el valor del parámetro que estamos introduciendo. A continuación, si ese valor es distinto de `cudaSuccess`, se escribe por pantalla que hay un error, y se escribe el tipo de error que da. Finalmente, en caso de que haya error, se sale del programa.

En definitiva, el código analiza si hay un error al haber hecho una cierta función (x). Si no lo hay, sigue igual, si lo hay, escribe por pantalla el tipo de error y sale del programa.

Ejercicio 3:

¿Por qué después de una llamada al device al lanzar un kernel no podemos usar el código de la macro que usamos siempre para buscar errores, sino que hay que hacer un `cudaGetLastError()` aparte?

Respuesta:

Porque, al contrario que en las funciones cuda como `cudaMemcpy()` o `cudaFree()`, los lanzamientos de un kernel no devuelves una variable de tipo `cudaError_t`, sino que no devuelven nada. Por ello en la variable x de la macro no obtendríamos nada, y hay que hacer un `cudaGetLastError()` aparte.

Ejercicio 4:

¿Qué es una FPU? ¿Cuál es la mayor diferencia entre una FPU y una GPU?

Respuesta:

FPU (*floating point unit*): coprocesador matemático antes de que surgieran las GPUs, utilizados para acelerar el procesamiento de datos. Diferencias respecto de las CPU: no pueden tener acceso a los datos directamente (debe ser la CPU la que gestione este apartado) o ejecutan un juego de instrucciones mucho más sencillo pensado para tratar datos en coma flotante.

Ejercicio 5:

¿Qué es la renderización?

Respuesta:

Es el proceso por el cual un modelo en 3D se transforma en una imagen o un vídeo.

Ejercicio 6:

¿A qué se refieren las palabras host y device? ¿Cuál es la relación entre ellos?

Respuesta:

La palabra host se refiere a la CPU, mientras que la palabra device se refiere a la GPU. Ambas trabajan en conjunto para computar de una forma eficiente. La CPU es la unidad central, es la que ejecuta los códigos y la que en contadas ocasiones llama a la GPU para que realice operaciones aritméticas donde esta unidad es más eficiente.

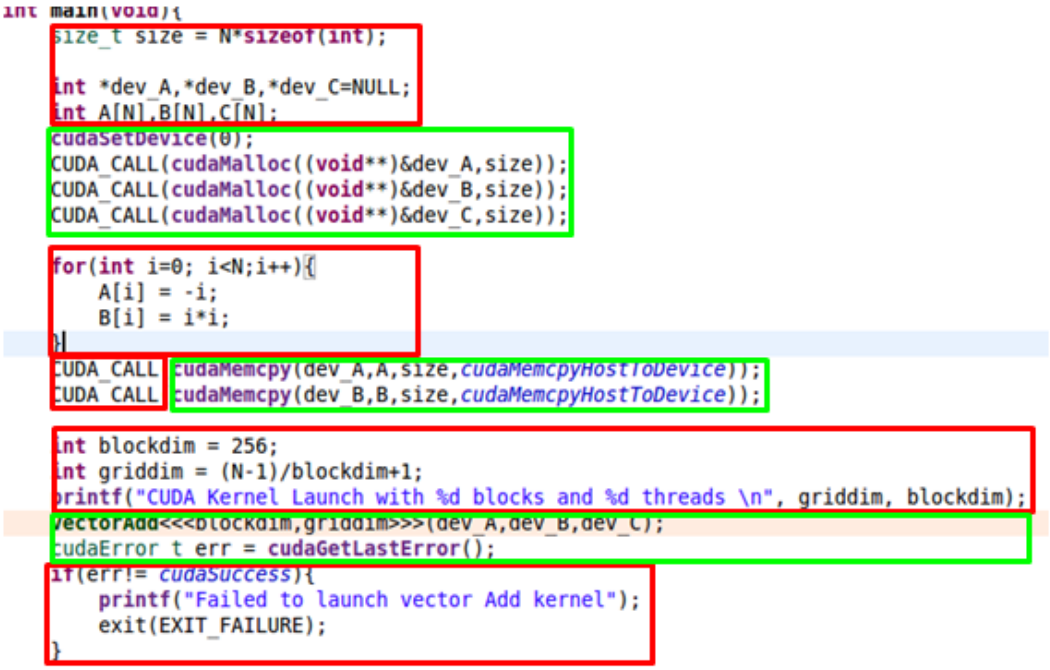
Ejercicio 7:

Sea el siguiente código.

```
int main(void){
    size_t size = N*sizeof(int);

    int *dev_A,*dev_B,*dev_C=NULL;
    int A[N],B[N],C[N];
    cudaSetDevice(0);
    CUDA_CALL(cudaMalloc((void**)&dev_A,size));
    CUDA_CALL(cudaMalloc((void**)&dev_B,size));
    CUDA_CALL(cudaMalloc((void**)&dev_C,size));

    for(int i=0; i<N;i++){
        A[i] = -i;
        B[i] = i*i;
    }
    CUDA_CALL(cudaMemcpy(dev_A,A,size,cudaMemcpyHostToDevice));
    CUDA_CALL(cudaMemcpy(dev_B,B,size,cudaMemcpyHostToDevice));

    int blockdim = 256;
    int griddim = (N-1)/blockdim+1;
    printf("CUDA Kernel Launch with %d blocks and %d threads \n", griddim, blockdim);
    vectorAdd<<<blockdim,griddim>>>(dev_A,dev_B,dev_C);
    cudaError_t err = cudaGetLastError();
    if(err!= cudaSuccess){
        printf("Failed to launch vector Add kernel");
        exit(EXIT_FAILURE);
    }
```

¿Qué instrucciones son realizadas por la CPU y que instrucciones son realizadas por la GPU? ¿Cómo funciona la comunicación entre *host* y *device*?

Respuesta:



En Rojo: Código ejecutado por el host.

En Verde: Código por ejecutado por el device.

La macro `CUDA_CALL()` no se ejecuta realmente en la GPU, ya que es en el preprocesamiento en el que esa macro queda sustituida por el código que ponemos siempre de comprobación de errores, y ese código es el que realmente se ejecuta en la CPU, por ello, esa línea de código, al expandirse, queda dividido en dos partes, una ejecutada por la CPU y otra ejecutada por la GPU.

El host se comunica con el device a través de llamadas o funciones. Dentro del código main, la CPU lanza un kernel con el programa que queremos que se ejecute en la GPU. Éste programa se ejecuta de forma más eficiente en la GPU, y se copian los resultados de nuevo al espacio de memoria situado en la CPU.

Ejercicio 8:

¿Para qué sirven los identificadores *BlockIdx* y *ThreadIdx*? ¿Qué datos tiene una variable de tipo dim3?

Respuesta:

Los identificadores *BlockIdx* y *ThreadIdx* son identificadores que nos indican tanto el bloque como el hilo con el que estamos trabajando. Cada bloque y cada hilo tienen un número distinto, y es perfecto para poder programar en paralelo. Ambos identificadores tienen tres variables, x y z, para cada una de las coordenadas en el caso de que nos queramos mover por el mapa de bloques.

Una variable de datos dim3 tiene 3 enteros, correspondientes a las tres coordenadas del eje x,y,z.

Se declara de la siguiente forma:

```
dim3  variable(ejeX,ejeY,ejeZ);
```

En el caso de que solo se declaren dos coordenadas, corresponderá al eje X y al eje Y, y el eje Z será un 1. Si solo se declara una coordenada, será el eje X, y el resto será un 1.

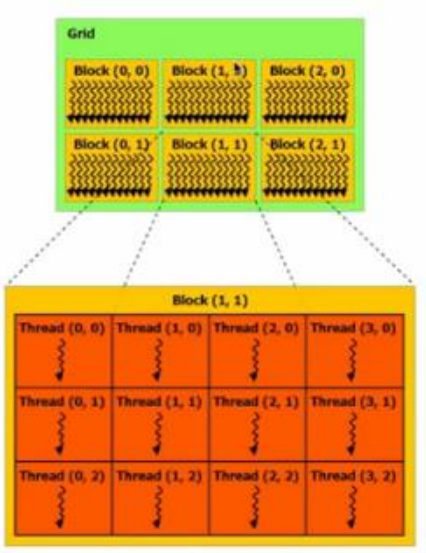
```
dim3 var(500,500,500); Matrix 500 x 500 x 500

dim3 var(500,500); Matrix 500 x 500 x 1

dim3 var(500); Matrix 500 x 1 x 1
```

Ejercicio 9:

¿Cómo se organiza un *grid*?

Respuesta:

Grid 2D: Hay un *grid* en el que hay distintos bloques, como es 2D, los bloques tienen dos coordenadas, una coordenada X y una coordenada Y. Los bloques comienzan desde (0,0) hasta (N-1, N-1). Dentro de los bloques, tenemos hilos o *threads*. Estos hilos se organizan de la misma forma que los bloques. En 2D, cada bloque tiene una cierta cantidad de hilos.



En el ejemplo de la figura, sería una gran distribución para manipular matrices de (2x3, 3x4) = (6,12) elementos.

Ejercicio 10:

¿Qué valores se tienen que pasar en la siguiente declaración?

```
Kernel_launch<<<valor1,valor2>>>(args,…);
```

Respuesta:

En la variable valor1, debemos introducir la dimensión del *grid*. En la variable valor2, debemos introducir la dimensión del bloque. En principio, serán variables dim3. Entre paréntesis pasaremos los argumentos que queremos que el kernel utilice para trabajar con ellos.