Universidad Politécnica
de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

Grado en ingeniería informática

Trabajo Fin de Grado

Paralelización y Vectorización de Redes Neuronales

Autor: Andrés Abelardo García Roqué

Tutor(a): Antonio García Dopico

Madrid, junio 2020

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática
Título: Paralelización y Vectorización de Redes Neuronales
Junio 2020

Autor: Andrés Abelardo García Roqué

Tutor:
Antonio García Dopico
DATSI
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

El objetivo de este proyecto es la paralelización y vectorización de una red neuronal para optimizar su tiempo de ejecución y explotar al máximo las capacidades de la máquina en la que se ejecuta. Para ello se realizará un análisis de los procesos de la red neuronal que más tiempo de ejecución consumen y se tratará de paralelizarlos y vectorizarlos para obtener incrementos de tiempo en la ejecución.

Previamente se traducirá la red neuronal escrita en Python al leguaje C para poder hacer uso de la librería OpenMP, la cual facilita mucho la ejecución de programas con múltiples procesos.

Abstract

The goal of the project is the parallelization and vectorization of a neural network to optimize its execution time and exploit the computer capabilities to the maximum extent possible. To achieve it an analysis of the neural network will be made to find which processes consume more time during the execution to try and parallelize and vectorize those processes and obtain gains in execution times.

Previously the neural network will be translated from Python to the C programming language to be able to use the OpenMP library which allows to execute multi-processing programs very easily.

Tabla de contenidos

1 Introducción

Las redes neuronales son cada día más predominantes en la ciencia de datos por su capacidad para obtener mejores resultados en problemas de clasificación que los algoritmos clásicos. Sin embargo, al componerse de un gran número de operaciones su ejecución puede llegar a ser muy lenta si no se aprovechan bien las capacidades de la máquina en la que se están ejecutando.

Esto es un problema típico en los lenguajes de más alto nivel en los que la sencillez de su manejo deja pocas posibilidades al programador para explotar las posibilidades de la máquina a bajos niveles. El ejemplo más típico es el lenguaje de programación Python, el cual ha ganado una tremenda popularidad en los últimos años por su sencillez y gran cantidad de librerías y es muy común para resolver problemas de ciencia de datos. Pero este lenguaje tiene desventajas respecto a otros lenguajes de más bajo nivel como C.

Python es un lenguaje interpretado, esto significa que no se compila si no que durante el tiempo de ejecución se evalúan los tipos de datos de las variables empleadas en el programa y no es necesario reservar memoria para los datos ya que esto se realiza de manera automática. Aunque para el programador estas propiedades resultan muy beneficiosas ya que le permiten escribir código de manera despreocupada sin declarar tipos de datos o gestionar la memoria, vienen acompañadas de un gran incremento en los tiempos de ejecución del programa debido a que se tienen que realizar reservas de memoria automáticas y comprobaciones de tipos de manera constante durante la ejecución.

En este proyecto se va a trabajar con una red neuronal escrita en Python, por lo que para obtener mejoras de rendimiento se traducirá a C para además poder realizar la paralelización y vectorización mediante el uso de la librería OpenMP [1].

La red neuronal que se va a utilizar es la empleada en el libro online “Neural Networks and Deep Learning” [2]. Esta red neuronal tiene como objetivo la clasificación automática de dígitos escritos a mano del dataset MNIST [3].

Una vez traducida la red neuronal se estudiará el consumo de tiempos de los distintos procesos de la red para analizar en que partes del código es más necesario aplicar la vectorización y paralelización para obtener las mayores mejoras de rendimiento.

Para poder aplicar las mejoras será necesario hacer un estudio de la librería OpenMP, para comprender su funcionamiento y para analizar las mejoras de rendimiento que produce sobre ejemplos de código aislados, principalmente sobre cálculos vectoriales como sumas o productos expresados dentro de bucles.

El objetivo final del proyecto es obtener la mayor mejora de rendimiento posible en la ejecución de la red neuronal teniendo en cuenta las limitaciones de la máquina en la que se ejecutará el código, además de aprender técnicas de paralelización y vectorización aplicables a otros tipos de programas.

2 Desarrollo

2.1 Introducción a las redes neuronales

En los subapartados de esta sección se explicarán de manera breve los conceptos básicos de una red neuronal y más concretamente de la red neuronal que se empleará durante el resto del proyecto, pero sin entrar en mucho detalle sobre el funcionamiento de las redes neuronales ya que el objetivo del proyecto no es crear una red neuronal si no mejorar el rendimiento de una ya existente.

2.1.1 Concepto de neurona

Durante los años 50 y 60 se definió el primer tipo de neurona artificial denominada perceptrón [4]. Este tipo de neuronas son las más simples pero son muy útiles para explicar el concepto de una neurona de manera sencilla.

Una neurona se compone de 4 elementos:

1. Valores de entrada (x_i)
2. Valor de salida (output)
3. Pesos (w_i)
4. Threshold (barrera de activación)

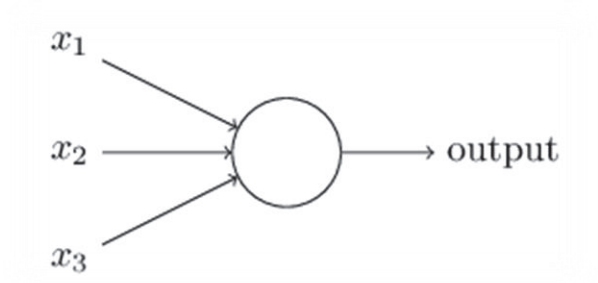


Ilustración 1 Perceptrón

Como se ve en la Ilustración 1, la neurona devuelve un output en función de una serie de parámetros de entrada.

Este output se calcula realizando el sumatorio del producto de cada una de las entradas con cada uno de los pesos que tiene asociado dicha entrada, si el resultado del sumatorio es menos o igual al threshold establecido la neurona devolverá un 0, en caso contrario devolverá un 1. En la Ilustración 2 se puede ver la expresión matemática que define lo anteriormente explicado.

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

Ilustración 2 Activación de un perceptrón

El problema con los perceptrones es que sus resultados son muy “estrictos”. Únicamente devolverán 0 o 1 como resultado y una vez que se alcanza la frontera entre el cambio de valor cualquier cambio en los datos que no vuelva a cruzar esa frontera resultará irrelevante, como se puede ver en la función de activación de un perceptrón en la Ilustración 3.

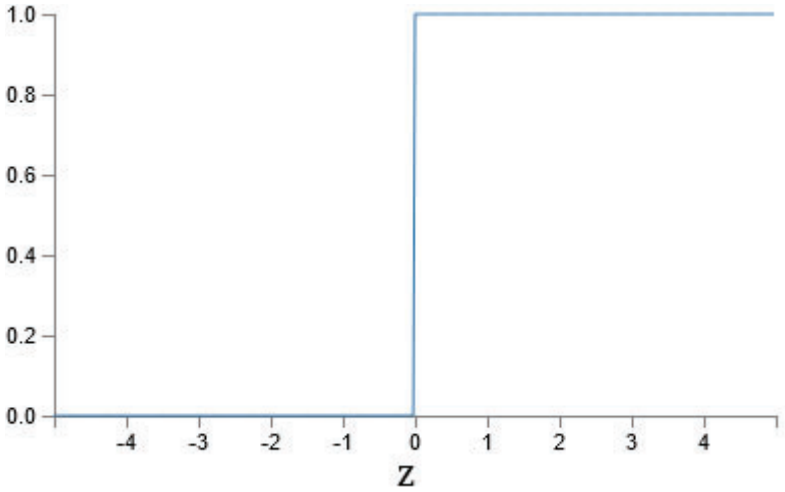


Ilustración 3 Función de activación de un perceptrón

Para poder obtener un rango de resultados más variable que 0 o 1 se estudiará otro tipo de neurona en el siguiente apartado llamadas neuronas sigmoides.

2.1.2 Neuronas sigmoides

Para poder obtener mayor variedad en los valores del resultado devuelto por la neurona se cambia la función de activación, en este caso se emplea la función logística, también conocida como función sigmoide representada en la Ilustración 4.

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}.$$

Ilustración 4 Función sigmoide

Esta fórmula nos permite obtener resultados distintos por pequeño que sea el cambio en z como se puede ver en la gráfica de la función de activación de la Ilustración 5.

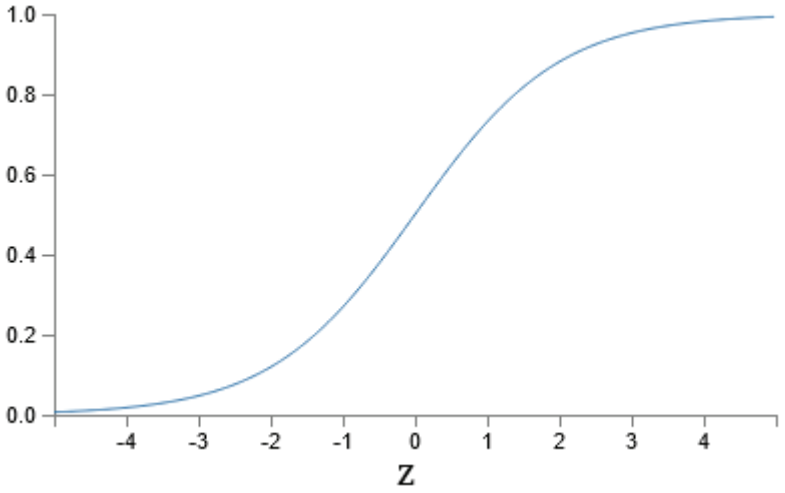


Ilustración 5 Función de activación de una neurona sigmoide

Estas son las neuronas que se van a emplear en la red neuronal del proyecto.

2.1.3 Arquitectura de redes neuronales

Una red neuronal se compone de múltiples neuronas agrupadas en capas de manera que la salida de las neuronas de una capa se pasa como la entrada de las neuronas de la siguiente capa.

Estas capas son:

- Capa de entrada (input layer)
- Capa de salida (output layer)
- Capas ocultas (hidden layers)

Las capas ocultas no son más que aquellas capas que no son ni de entrada ni de salida como se puede ver en la Ilustración 6.

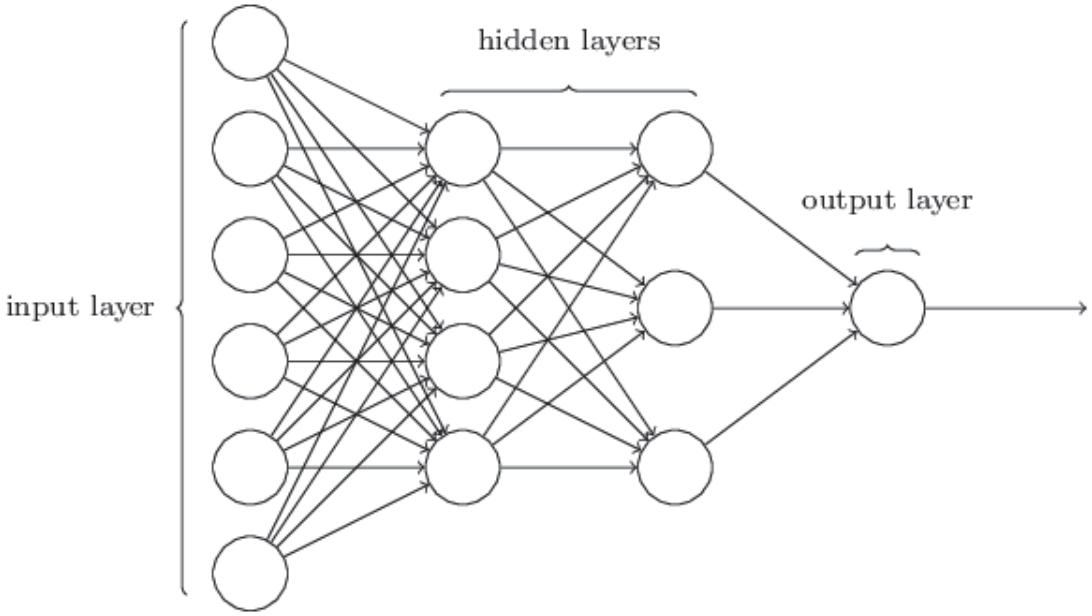


Ilustración 6 Arquitectura de una red neuronal

El número de neuronas en las capas de entrada y de salida es completamente dependiente del problema que se pretenda resolver. Como se puede ver en la Ilustración 6 la red neuronal tiene dos capas intermedias, la primera con cuatro neuronas y la siguiente con 3. El número de capas intermedias la distribución del número de neuronas por capa es un problema difícil de resolver aunque en los casos más simples existen reglas que facilitan las decisiones.

Cuando las neuronas están totalmente conectadas, es decir, las salidas de una capa son las entradas de todas las neuronas de la capa siguiente se dice que tiene capas densas. A las redes neuronales que se componen únicamente de capas densas se las llama *Feed Forward Neural Network* o *Fully Connected Neural Network.* Son estructuras sencillas y es el tipo de red que se va a utilizar en este proyecto.

2.1.4 Red neuronal MNIST

El objetivo de la red neuronal con la que se va a trabajar es reconocer dígitos del 0 al 9 escritos a mano. Cada uno de estos dígitos es una imagen en escala de grises 0-255 compuesta por 28*28 píxeles. En la Ilustración 7 se ve un ejemplo de estos dígitos.

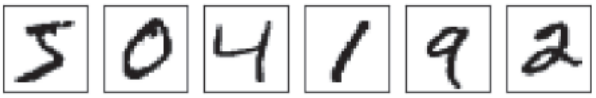


Ilustración 7 Ejemplos de imágenes de MNIST

Los datos con los que se va a trabajar consisten en una dupla de 784 números (28*28) correspondientes a los píxeles de las imágenes y un valor indicando cual es el dígito asociado a esa imagen.

Como las imágenes se componen de 784 píxeles este será el número de entradas de la capa *input* de la red neuronal, y al ser dígitos del 0 al 9 tendremos en la capa *output* 10 neuronas representando cada una uno de los dígitos, activándose la neurona que esté en la posición del dígito.

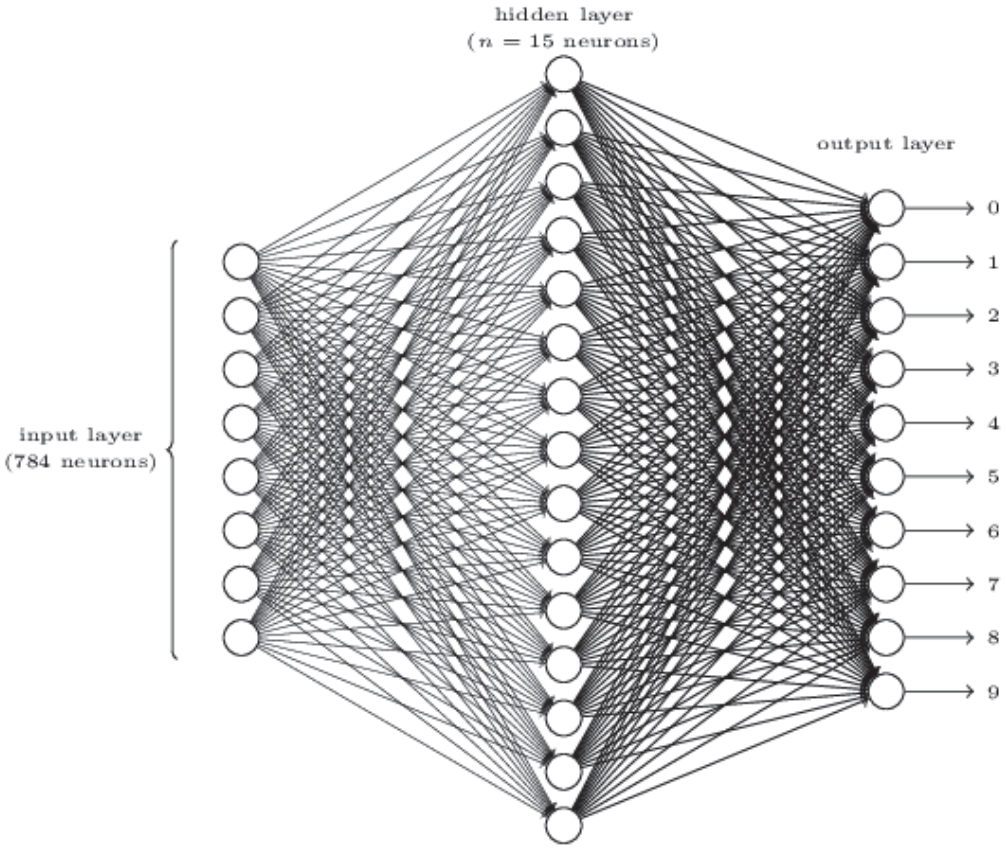


Ilustración 8 Esquema de la red neuronal MNIST

En la Ilustración 8 se puede ver una representación esquemática de la red neuronal con la que se va a trabajar. La capa de entrada tiene el número de neuronas reducido para poder visualizarla mejor. Se muestra una única capa oculta compuesta por 15 neuronas pero este valor puede variar a lo largo del uso desarrollo del proyecto y únicamente se muestra para visualizar de forma completa la red.

2.2 Vectorización

En este apartado se va a explicar el concepto de vectorización de manera teórica y práctica, mediante ejemplos con código y herramientas para vectorizar de manera automática y manual además de como aplicar esta vectorización a la red neuronal con la que se trabaja.

2.2.1 Introducción a la vectorización

La vectorización es una manera de explotar el paralelismo a nivel de datos aplicando la misma operación sobre múltiples elementos de datos de manera paralela.

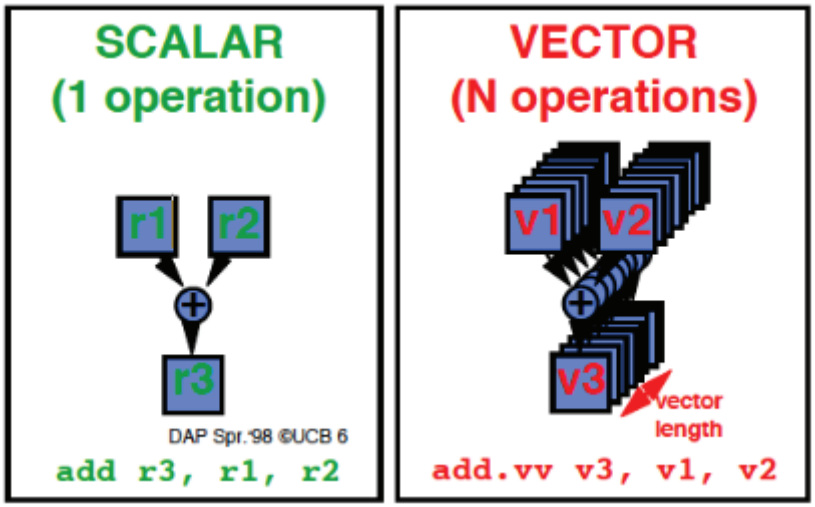


Ilustración 9 Esquema de vectorización [5]

En la Ilustración 9 se puede ver como una operación escalar típica únicamente se aplica sobre un elemento mientras que una operación vectorizada es capaz de hacer esa operación escalar *n* veces de manera simultánea, arrojando los resultados sobre un vector de tamaño *n*.

Para poder vectorizar es necesario modificar el programa de manera que una única instrucción pueda ejecutar múltiples operaciones en distintos datos.

La vectorización se aplica comúnmente a operaciones vectoriales dentro de bucles en los cuales se realiza una operación por cada elemento del bucle.

2.2.2 Requisitos para vectorizar

Aunque es necesario hacer modificaciones en el software, la vectorización no es posible sin un hardware apropiado. Es imprescindible disponer de unidades vectoriales en nuestra máquina.

Hoy en día las unidades vectoriales más comunes se encuentran en las tarjetas gráficas de los ordenadores, cuya principal tarea es realizar cálculos aritméticos y al emplear sus capacidades vectoriales aceleran estos cálculos enormemente, pero además esto abre una posibilidad de aprovechar este hardware para acelerar los cálculos de la red neuronal.

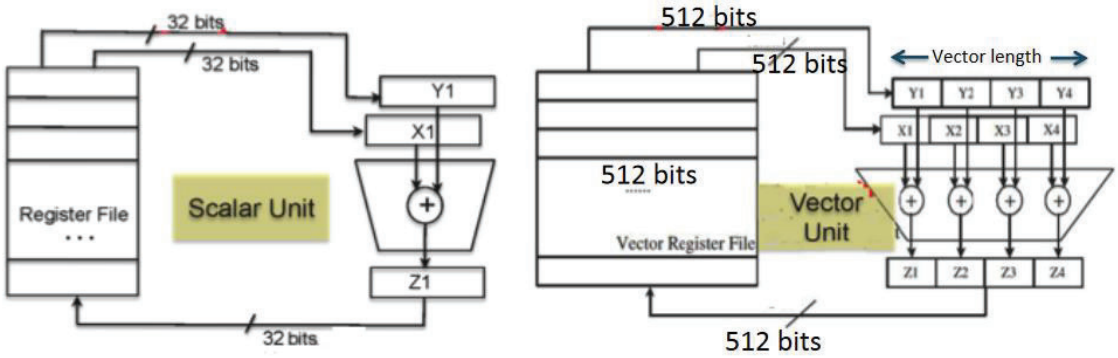


Ilustración 10 Unidad escalar y unidad vectorial

2.2.3 ¿Por qué vectorizar?

Vectorizar un programa de manera manual (más adelante se estudiará como hacerlo de forma automática) conlleva un esfuerzo adicional a la hora de escribir el código, pero tiene muchas más ventajas.

El acceso a memoria de las instrucciones vectoriales se hace mediante un patrón conocido por lo que se reducen mucho los tiempos de búsqueda en memoria. Para poder realizar una operación vectorial se debe alinear en memoria dicho vector de forma que los datos estarán almacenados de manera consecutiva, por lo tanto, no será necesario buscar la información en distintas partes de la memoria si no que un dato siempre estará a continuación del anterior en la memoria, lo cual reduce mucho los tiempos de búsqueda.

Cada instrucción trae muchos más datos y realiza más trabajo por lo que resulta más eficiente energéticamente al tener que ejecutarse menos instrucciones (alrededor de 10x instrucciones menos [5]).

Según avanza la tecnología se obtienen recursos más poderosos para realizar operaciones vectoriales por lo que todo el proceso es escalable. Se estima que el tamaño de los vectores hardware se duplicará cada cuatro años.

2.2.4 Rendimiento de la vectorización

El rendimiento es sensible a varias condiciones que se deben tener muy en cuenta si se quieren obtener ganancias.

-Longitud del vector en la unidad hardware:

El rendimiento se verá directamente afectado por la longitud que nuestro hardware nos permitirá vectorizar, cuanto más grande sea el tamaño más se podrá aumentar el rendimiento.

-Los datos deben estar alineados en caché:

Previamente se ha mencionado el alineamiento en memoria como una ventaja de la vectorización ya que reduce los tiempos de búsqueda en memoria, pero hay que tener en cuenta que se debe alinear esta memoria previamente y ajustarla al tamaño de los bloques de caché. Si los datos no están alineados en memoria se necesitarán más instrucciones para recogerlos y organizarlos de manera que se puedan realizar operaciones SIMD. El movimiento de datos es óptimo si la dirección de los datos comienza en ciertos bytes (16, 32 o 64 dependiendo de la arquitectura del computador).

-Porcentaje del código que se puede vectorizar y tamaño de los vectores dentro de los bucles:

Para obtener una ganancia real al vectorizar un programa es necesario que dicho programa realice suficientes operaciones sobre vectores ya que si no es así el incremento de ganancia no será a penas apreciable. En el caso de las redes neuronales son candidatas excelentes a ser vectorizadas puesto que gran parte de sus operaciones consisten en sumas y productos vectoriales. El tamaño de los vectores con los que se va a operar dentro de los bucles es muy importante ya que si no son lo bastante grandes como para utilizar todas las capacidades de la unidad vectorial se perderá ganancia.

-Coste computacional de las operaciones sobre los vectores:

La ganancia se verá afectada dependiendo de la complejidad de las operaciones que queramos aplicar a los vectores. Las operaciones aritméticas más sencillas (suma, resta, multiplicación y división) se realizarán muy rápido, pero operaciones más complejas pueden aumentar el coste y anular la posibilidad de vectorizar.

-Situaciones especiales:

Más adelante se estudiarán situaciones especiales en más detalle, pero entre ellas se encuentra una muy común; condiciones dentro del bucle que impiden que se realice la operación deseada en todas las iteraciones, por lo que la ganancia dependerá del número de iteraciones en las que sí se realiza la operación.

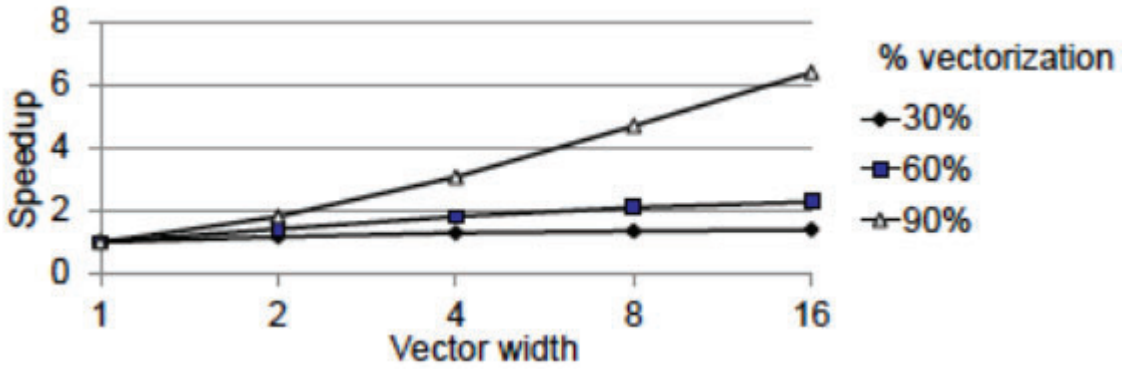


Ilustración 11 Ganancia de tiempo de ejecución frente al ancho de un vector

Según la ley de Amdahl "*La mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente.*"

Esto significa en nuestro caso que la mejora de rendimiento que seremos capaces de obtener mediante la vectorización está limitada por el tiempo que nuestro programa gasta realizando operaciones vectoriales. Por mucho que ajustemos jamás llegaremos a una ganancia del 100% ya que se deben ejecutar instrucciones que no son vectorizables en alguna parte del código.

Existe una manera adicional de mejorar el rendimiento de la vectorización facilitando la organización y carga de datos de manera que estén alineados pero resulta muy poco intuitiva a la hora de programar, se trata de la diferencia entre un *array de estructuras* y una *estructura de arrays*.

Un array de estructuras es la manera natural de organizar los datos pero deja vectores iguales en posiciones de memoria no consecutivas de manera que hay que reorganizarlos para poder alinearlos. A continuación se ilustra con un ejemplo.

Supongamos que disponemos de una estructura definida en C que se compone de 3 arrays distintos representados en la Ilustración 12.

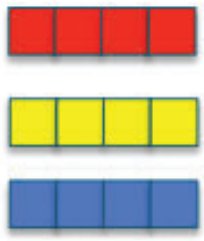


Ilustración 12 Estructura formada por 3 arrays

Si declaramos un array formado por esta estructura en memoria se alineará de manera que haya un vector rojo, luego uno amarillo y por último uno azul, repitiendo este patrón tantas veces como elementos queramos que tenga nuestro array de estructuras como se puede ver en la Ilustración 13 (en esta ilustración cada array está representado por un único cuadrado para facilitar la visualización).

Ilustración 13 Array de estructuras

Es fácil ver que si se pretende operar de manera conjunta con los vectores de cualquier color se deberán extraer individualmente ya que no están alineados en memoria.

Para acelerar este proceso se puede plantear el problema como una estructura de arrays. En lugar de tener una cadena de estructuras con arrays de tamaños reducidos se crea una única estructura con arrays del tamaño completo necesario y se rellena de manera que los datos queden alineados en memoria como se puede ver en la Ilustración 14.

Ilustración 14 Estructura de arrays

Aunque esta técnica facilita mucho las operaciones vectoriales al reducir en gran medida las instrucciones necesarias para alinear la memoria, tiene como desventaja que es muy poco intuitiva tanto para el programador como para quienes tengan que leer e interpretar el código posteriormente. Además complica la modificación de los datos de una parte de los arrays ya que se debe acceder a la posición de memoria correcta y si no se hace bien puede causar errores.

2.2.5 Como vectorizar código

Aunque existen varias formas de vectorizar un código en este proyecto solo se va a estudiar dos de ellas. Vectorización mediante el uso de la librería OpenMP y vectorización automática guiada por anotaciones mediante el compilador.

2.2.5.1 Vectorización mediante OpenMP

A partir de la versión 4.0 y en versiones posteriores [6] de OpenMP es posible vectorizar código empleando directivas SIMD (Single-Instruction Multiple-Data).

La sintaxis de una directiva SIMD es la siguiente:

```
#pragma omp simd[clause[[,] clause]…]
      //Bucle for
```

Donde *clause* es una de las siguientes:

- `if([simd :] scalar-expression)`
- `safelen(length)`
- `simdlen(length)`
- `linear(list[ : linear-step])`
- `aligned(list[ : alignment])`
- `nontemporal(list)`
- `private(list)`
- `lastprivate([ lastprivate-modifier:] list)`
- `reduction([ reduction-modifier,]reduction-identifier : list)`
- `collapse(n)`
- `order(concurrent)`

2.2.5.1.1 Producto de dos vectores

El siguiente código muestra el producto de dos vectores *b* y *c*, almacenando el resultado en un tercer vector *a*.

```
#pragma omp simd aligned(a : 64)
for(i=0; i<MAX; i++){
      a[i] = b[i] * c[i];
}
```

Mediante la primera línea (marcada en rojo) indicamos que el bucle puede ser transformado en un bucle SIMD, lo que significa que múltiples iteraciones del bucle pueden ser ejecutadas de forma concurrente utilizando instrucciones SIMD. La cláusula `aligned(a : 64)` indica que los datos del vector *a* están alineados en memoria, siendo 64 el ancho del vector.

Para reservar memoria de manera que esté alineada se emplean variantes de la función `malloc()` cuya sintaxis es distinta dependiendo del sistema en el que trabajemos.

A continuación se muestra un ejemplo para arquitecuras Intel

```
      _mm_malloc(8*sizeof(float), 64);
```

Mediante la llamada a función del ejemplo se realiza una reserva de 8 elementos (en posiciones consecutivas de memoria) de tipo *float* indicando un ancho de vector de 64 bytes. Esta reserva de memoria es válida para arquitecturas Intel pero hay más opciones válidas como las que se muestran a continuación:

C11:

```
void * aligned_alloc (size_t alignment, size_t size)
```

POSIX:

```
int posix_memalign (void **memptr, size_t alignment, size_t size)
```

Windows:

```
void * _aligned_malloc(size_t size, size_t alignment);
```

Aunque es importante reservar la memoria de manera alineada cabe destacar que para vectores relativamente pequeños y especialmente en sistemas modernos con mucha memoria no es necesario especificar que se alineen los datos memoria ya que el vector siempre se almacenará de manera consecutiva al no ocupar demasiado espacio

Para probar los resultados se han multiplicado dos vectores de 100000 elementos de tipo *double* mediante dos funciones, una vectorizada y otra no. Estas operaciones se han repetido 100000 veces midiendo los tiempos de cada una de ellas y calculando la media total obteniéndose los siguientes resultados:

- La versión vectorizada es 1.340402 veces más rápida de media.
- La mayor ganancia ha sido 19.304134 veces más rápida la versión vectorizada, con un tiempo de ejecución de 0.000906 segundos en la versión normal y 0.000047 segundos en la versión vectorizada.

NOTA: Cuando se compila el programa utilizando el compilador GCC es muy importante desactivar la vectorización automática que realiza el compilador, la cual se estudiará mas adelante. Para desactivarla basta con añadir las opciones `-O3 -fno-tree-vectorize`.

2.2.5.1.2 Suma acumulada de valores

El primer ejemplo es extrapolable a bucles iguales en los que únicamente cambie la operación por una suma, una resta o una división.

Si queremos realizar operaciones un poco más complejas dentro de nuestros bucles será necesario indicar en las directivas SIMD las clausulas especiales para que se ejecute el código correctamente de manera vectorial.

El ejemplo que se muestra a continuación realiza el mismo cálculo que el ejemplo anterior y además se realiza la suma acumulada de los resultados obtenidos en *a* guardando el valor en la variable *sum*. Si ejecutásemos este código con la cláusula SIMD anterior no se realizaría de manera correcta la acumulación de valores sobre la variable *sum* ya que es un cálculo simultáneo y se almacenaría únicamente uno de los valores de la suma, un error parecido a una condición de carrera [7] en problemas de concurrencia, por lo que se debe modificar.

```
#pragma omp simd reduction(+:sum) aligned(a : 64)
for(i=0; i<MAX; i++){
      a[i] = b[i] * c[i];
      sum = sum + a[i];
}
```

Para solucionar el problema se añade `reduction(+:sum)` a la directiva SIMD. De esta forma se crearán copias de nuestra variable sum, al final de cada región SIMD que corresponde a la directiva sobre la que se ha especificado la cláusula *reduction,* la variable original se actualiza combinando el valor original con el

valor final de cada una de las copias, usando como combinación la operación especificada en la cláusula, en este caso una suma.

Aplicando el mismo número de iteraciones con los dos mismos vectores que en el ejemplo anterior se han obtenido los siguientes resultados:

- La versión vectorizada es 1.360772 veces más rápida de media.
- La mayor ganancia ha sido 12.552711 veces más rápida la versión vectorizada, con un tiempo de ejecución de 0.001048 segundos en la versión normal y 0.000083 segundos en la versión vectorizada.

Para este ejemplo es esencial saber si el resultado de ambas sumas acumuladas es el mismo, ya que si no lo fuesen resultaría irrelevante obtener mejorías en tiempos de ejecución debido a que los resultados serían erróneos.

La versión del código no vectorizada devuelve un valor en la suma acumulada de 333328333350000.00 y este resultado es el mismo en la versión vectorizada por lo que estas ganancias de tiempo son válidas al ser ambos resultados iguales.

2.2.5.1.3 Declaración de funciones SIMD

OpenMP también permite la declaración de funciones SIMD. Estas funciones resultan útiles cuando se desea realizar una combinación de varias operaciones de manera vectorial.

A continuación se muestra un ejemplo:

```
#pragma omp declare simd
int miFuncion(int a, int b, int c){
      return a*b+c;
}
```

Se utiliza la cláusula *declare* en la directiva SIMD para indicar que es una función vectorial que posteriormente se utilizará dentro de un bucle SIMD como se muestra a continuación:

```
#pragma omp simd
for(i=0; i<MAX; i++){
      resultado[i] = miFuncion(vectorA[i], vectorB[i], vectorC[i]);
}
```

Aplicando el mismo número de iteraciones con los dos mismos vectores que en el ejemplo anterior se han obtenido los siguientes resultados:

- La versión vectorizada es 1.103815 veces más rápida de media.
- La mayor ganancia ha sido 159.205329 veces más rápida la versión vectorizada, con un tiempo de ejecución de 0.001272 segundos en la versión normal y 0.000008 segundos en la versión vectorizada.

Aunque el valor máximo de ganancia sea muy elevado la media es muy baja y los valores máximos oscilan entre 50 y 200 si se repite la prueba, por lo que no es una medida consistente.

A pesar de las muchas ventajas que ofrece la vectorización mediante OpenMP debe tratarse con cuidado. Utilizar OpenMP SIMD se salta los análisis del compilador para vectorizar de manera automática y hoy en día la vectorización

automática puede llegar a ser muy potente, especialmente en compiladores como GCC.

Se debe utilizar OpenMP con precaución, se pueden obtener resultados incorrectos, errores de memoria e incluso empeoramiento en los tiempos de ejecución, así que es esencial revisar a fondo y comprender lo que se hace para no arruinar el código que se pretende mejorar.

2.2.5.2 Vectorización automática mediante compiladores

En esta sección se va a tratar la vectorización automática. Concretamente la realizada por el compilador GCC sobre el lenguaje C.

GCC es capaz de vectorizar de forma automática secciones de código que cumplan ciertas condiciones. Para facilitar esta tarea se pueden emplear directivas o anotaciones que sugieren al compilador que partes del código pueden ser vectorizadas, y programar de manera que resulte más fácil reconocer estas secciones.

Para activar la vectorización en GCC se debe activar el flag `-ftree-vectorize` pero no es necesario si se usa la opción de optimización `-O3` ya que viene por defecto dentro de esta. Para obtener un informe sobre las partes del código que han sido vectorizadas se puede usar el flag `-ftree-vectorizer-verbose.`

También se pueden utilizar los siguientes flags al compilar en GCC para obtener información adicional sobre la vectorización realizada por el compilador:

- `-fopt-info-vec` o `-fopt-info-vec-optimized` con ellos el compilador indicará el número de línea de los bucles que han sido optimizados para vectorización.
- `-fopt-info-vec-missed` información detallada sobre los bucles que no han sido optimizados.
- `-fopt-info-vec-note` información detallada sobre los bucles y vectorizaciones que se realizan.
- `-fopt-info-vec-all` todas las opciones anteriores combinadas.
- `-march=native` utiliza instrucciones válidas para la CPU en la que se ejecuta.
- `-falign-functions=`N alinea las direcciones de las funciones para que sean múltiplo de N bytes (los valores de N suelen ser potencias de 2: 8, 16, 32, 64…).
- `-falign-loops=`N alinea las direcciones de los bucles para que sean múltiplo de N bytes (los valores de N suelen ser potencias de 2: 8, 16, 32, 64…).
- Para permitir vectorización de reducciones de coma flotante se usan los flags `-ffast-math` o `-fassociative-math`.

A continuación se va a probar la vectorización automática sobre un ejemplo sencillo, el producto de dos vectores.

```
for(i=0; i<MAX; i++){
      a[i] = b[i] * c[i];
}
```

Compilando el programa con GCC y usando los flags -O2 `-ftree-vectorize -fopt-info-vec` obtenemos lo Ilustración 13.

```
vectorizacion_automatica.c:14:5: note: loop vectorized
vectorizacion_automatica.c:31:2: note: loop vectorized
```

Ilustración 15 Compilación mediante flags de vectorización 1

Los logs de la compilación indican que se han vectorizado dos bucles, uno en la línea 14 y otro en la 31. El de la línea 14 es el bucle que realiza el cálculo que deseamos mientras que el de la línea 31 es un bucle auxiliar utilizado para rellenar los vectores con valores.

Tras multiplicar dos vectores de 100000 elementos durante 10000 iteraciones obtenemos que el tiempo de ejecución es 0.327976 segundos.

Tras ejecutar el mismo cófigo pero sin indicar al compilador que realice vectorizaciones automáticas se obtiene un tiempo de ejecución de 2.113203 segundos, obteniéndose una ganancia de 6.44 en la versión vectorizada con *flags* frente a la no vectorizada.

Por último se ha compilado el programa con el *flag* -O3 que habilita la máxima optimización disponible por el compilador, incluyendo vectorización. Obteniendo un tiempo de ejecución de 0.292427 segundos, lo que representa una ganancia de 1,12 frente a la vectorización indicada mediante flags.

Al igual que en el caso de vectorización mediante OpenMP también se va a analizar el resultado de vectorizar un bucle en el que se realiza una suma acumulada, además del previo producto de dos vectores.

```
for(i=0; i<MAX; i++){
      a[i] = b[i] * c[i];
      sum = sum + a[i];
}
```

Compilando el programa con GCC y usando los flags -O2 `-ftree-vectorize -fopt-info-vec` obtenemos lo siguiente:

```
vectorizacion_automatica.c:15:5: note: loop vectorized
vectorizacion_automatica.c:32:2: note: loop vectorized
```

Ilustración 16 Compilación mediante flags de vectorización 2

Como se puede ver, la Ilustración 16 es muy similar a la Ilustración 15 ya que el código que se compila es casi idéntico, únicamente aumenta una línea la posición en la que se encuentran los bucles ya que se ha añadido una línea de código para realizar la suma acumulada del vector.

Realizando las mismas iteraciones que en el ejemplo anterior se ha obtenido un tiempo de ejecución de 0.314848 segundos.

Al realizar la compilación sin fla*flags* adicionales para evitar que realice cualquier tipo de vectorización se obtiene un tiempo de ejecución de 2.139521 segundos. Esto representa una ganancia de 6.79.

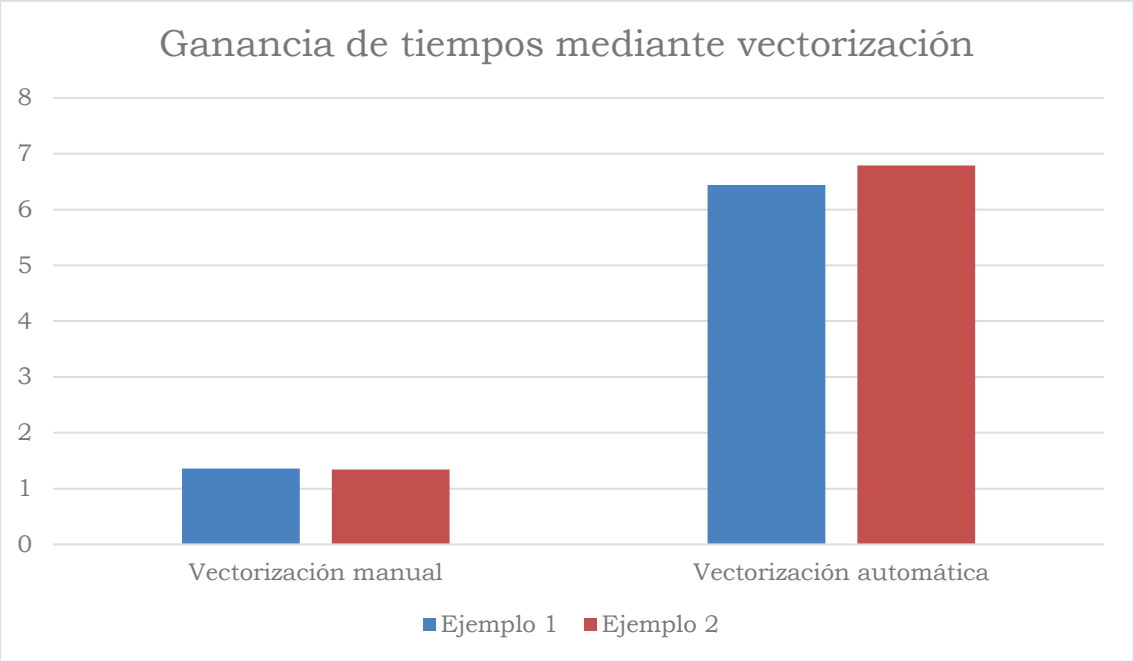
Por último y como en el ejemplo anterior se ha compilado el código con la opción -O3 para obtener la máxima optimización posible por el compilador, obteniendo un tiempo de ejecución de 0.286738 segundos, lo que representa una ganancia de 1.09 respecto a la vectorización indicada mediante flags.

2.2.5.3 Resultados y conclusiones

La vectorización automática es más sensible que la vectorización manual, y si no se cumplen una serie de requisitos el compilador no vectorizará los bucles. Entre esos requisitos los que se exponen a continuación son los más comunes:

- Hay que asegurarse de que el número de iteraciones es fijo al entrar en el bucle, es decir, no se puede alterar de manera dinámica la variable que limita las iteraciones dentro del bucle.
- El bucle debe tener una única entrada y una única salida.
- Se debe evitar el uso de sentencias `break` o `continue` en la medida de lo posible, a veces el compilador es capaz de vectorizar el bucle pero en algunos casos no será posible.
- No debe haber llamadas a funciones dentro del bucle o deben ser muy limitadas. Las llamadas a funciones matemáticas vectorizables están permitidas.
- No debe haber dependencias de datos dentro del bucle con otros índices de los vectores, por ejemplo, una referencia de este estilo "`vector[i-1]`" depende de un valor anterior a la iteración "i" en la que nos encontremos, por lo que no se va a obtener ya que se calcula de manera simultánea.
- Las sentencias condicionales (`if/else`) pueden ser utilizadas siempre y cuando no cambien el flujo de control y solo son utilizadas para cargar valores A o B en una variable C. Es decir, si una sentencia condicional interrumpe la asignación de valores en un vector provocará que el bucle no sea vectorizable.

Aunque parezca que son una gran cantidad de condiciones las que hay que cumplir para poder obtener ganancias reales de la vectorización automática, lo cierto es que hay que cumplir prácticamente todas esas condiciones cuando se vectoriza empleando directivas SIMD de OpenMP con la desventaja de que usando OpenMP hay que estar atento a todas las posibles opciones para vectorizar, mientras que el compilador lo intentará hacer siempre que vea la oportunidad.



Viendo los resultados obtenidos en los que la ganancia obtenida en la vectorización automática llega a ser mucho más elevada que en la

vectorización manual es más recomendable emplear el sistema que ofrece el compilador. Cabe destacar que la ganancia obtenida mediante vectorización por el compilador frente a la obtenida con el *flag* -O3 (optimización completa) es muy parecida, esto se debe principalmente a que se han tratado ejemplos muy sencillos en los que casi todo el poder de procesamiento estaba concentrado en operaciones entre vectores, pero en problemas reales la optimización completa `-O3` puede obtener mucha más ganancia.

Sin embargo la vectorización automática puede no ser perfecta y en los casos en los que no se vectoricen bucles de manera automática se deberá hacer un análisis para intentar detectar y corregir las razones por las que no se ha vectorizado automáticamente, o directamente vectorizarlo a mano empleando OpneMP.

2.3 Paralelización

En este apartado se va a estudiar las ventajas que ofrece la paralelización en cuanto a ganancias en tiempo de ejecución, tratando de explotar al máximo las capacidades de la máquina en la que se ejecuta el código. Para ello se va a emplear la librería OpenMP que ya se estudió en la sección anterior.

2.3.1 Introducción a la paralelización

La paralelización consiste en la ejecución simultánea y coordinación de procesos con un mismo objetivo final en sistemas de memoria compartida.

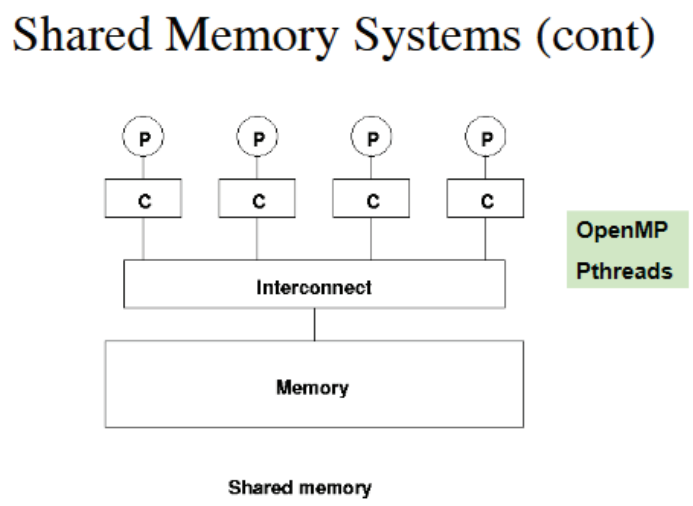


Ilustración 17 Sistema de memoria compartida

Como se ve en la Ilustración 17 un sistema de memoria compartida consiste en un conjunto de procesadores (p) que son capaces de trabajar de manera independiente, compartiendo una memoria común.

La gran mayoría de los ordenadores modernos disponen de capacidades de multiprocesamiento, pero muchas veces no se explotan al máximo por parte de los desarrolladores software, ya que la sincronización de los procesos y del acceso a memoria puede llegar a ser una tarea complicada.

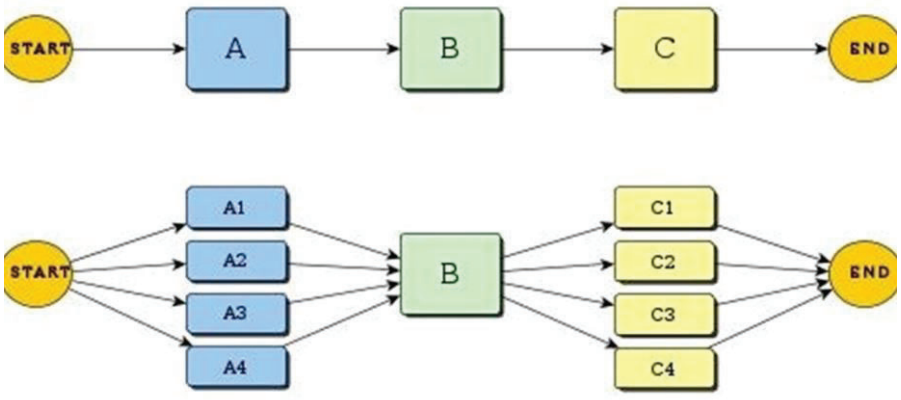


Ilustración 18 Diagrama de ejecución paralela

En la Ilustración 18 se presenta un diagrama de una ejecución paralela de código. Las secciones A y C se denominan regiones paralelas. Son secciones de código que se pueden ejecutar mediante múltiples procesadores de forma simultánea sin que esto afecte al resultado final. La sección B es una región no paralelizable. Estas regiones son partes del código que o bien no se pueden paralelizar debido a que se alteraría el resultado si se hiciese, o bien son demasiado pequeñas como para que merezca la pena ser paralelizadas.

La existencia de estas regiones no paralelizables impiden que se alcance un 100% de ganancia según la ley de Amdhal: "*La mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente.*".

2.3.2 Requisitos para paralelizar

Es imprescindible disponer de un hardware con capacidad de multiprocesamiento y sistemas de memoria compartida como el mostrado en la Ilustración 17. Esta memoria no viene asociada directamente con la cantidad de procesadores que dispone la máquina por lo que el sistema no es tan fácilmente escalable como los sistemas de memoria distribuida que se verán a continuación.

La ventaja que ofrecen estos sistemas es que son mucho más comprensibles por el usuario. Al usar un sistema de memoria compartido y accesible de igual manera por todos los procesadores el programador puede entender y organizar mejor el código que pretende escribir. Además, el acceso a los datos es mucho más rápido al estar distribuidos en posiciones próximas de memoria.

También son válidos, aunque menos extendidos, los sistemas de memoria distribuida como el mostrado en la Ilustración 19. En estos sistemas cada procesador dispone de su propia memoria de manera que si se quiere compartir información entre procesadores un procesador debe acceder a la memoria de otro procesador.

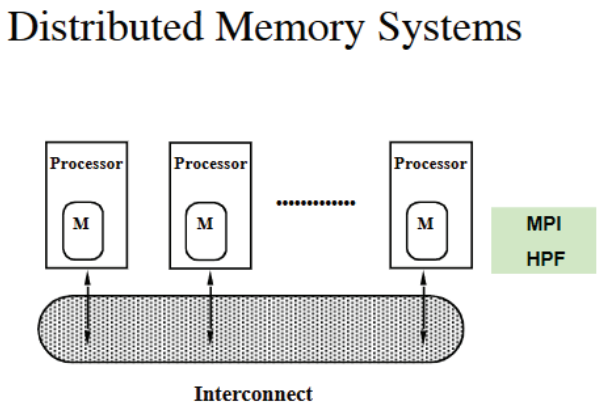


Ilustración 19 Sistema de memoria distribuida

La principal ventaja que ofrecen los sistemas de memoria distribuida es que la memoria escala con el número de procesadores, es decir si aumentamos el número de procesadores también aumentará la memoria del sistema.

Estos procesadores pueden acceder muy rápidamente a los datos al disponer de memorias independientes, pero cuando se pretende acceder a la memoria de otros procesadores se complica y ralentiza mucho el acceso. También resulta

complicado mapear estructuras de datos comunes de gran tamaño ya que no se dispone de una memoria global y el tiempo de acceso a los datos se vuelve no uniforme ya que cada procesador tardará un tiempo distinto en acceder a los datos de su memoria.

2.3.3 ¿Por qué paralelizar?

Las razones para paralelizar código resultan más obvias que en la vectorización. Normalmente es más conocido que los ordenadores disponen de múltiples procesadores y tienen capacidad para realizar múltiples tareas a la vez, a pesar de esto, a la hora de programar no se tienen en cuenta estas capacidades o no se aprovechan por pensar que resulta complicado gestionar múltiples procesos. Sin embargo, gracias a librerías como OpneMP la paralelización es mucho más accesible para los usuarios sin las preocupaciones de gestión de múltiples procesos o evitar condiciones de carrera, interbloqueos o desincronización.

En cuanto al rendimiento, el tiempo de ejecución de un programa en situaciones ideales es inversamente proporcional al número de procesadores que lo ejecuten, es decir, si un programa se ejecuta de manera simultánea en cuatro procesadores el tiempo de ejecución será un cuarto del tiempo que tarde en ejecutarse en un único procesador, más adelante se verá que esto no es así ya que es imposible alcanzar situaciones ideales en las que todo el código se ejecute de manera paralela.

2.3.4 Rendimiento de la paralelización

Como se ha mencionado en secciones anteriores nunca se podrá alcanzar un incremento de rendimiento total debido a la ley de Amdhal, al no pasar la totalidad del tiempo del programa en regiones paralelizables, pero para poder explotar al máximo el rendimiento dentro de dichas regiones se pueden seguir una serie de técnicas para mejorar la ejecución y los accesos a memoria.

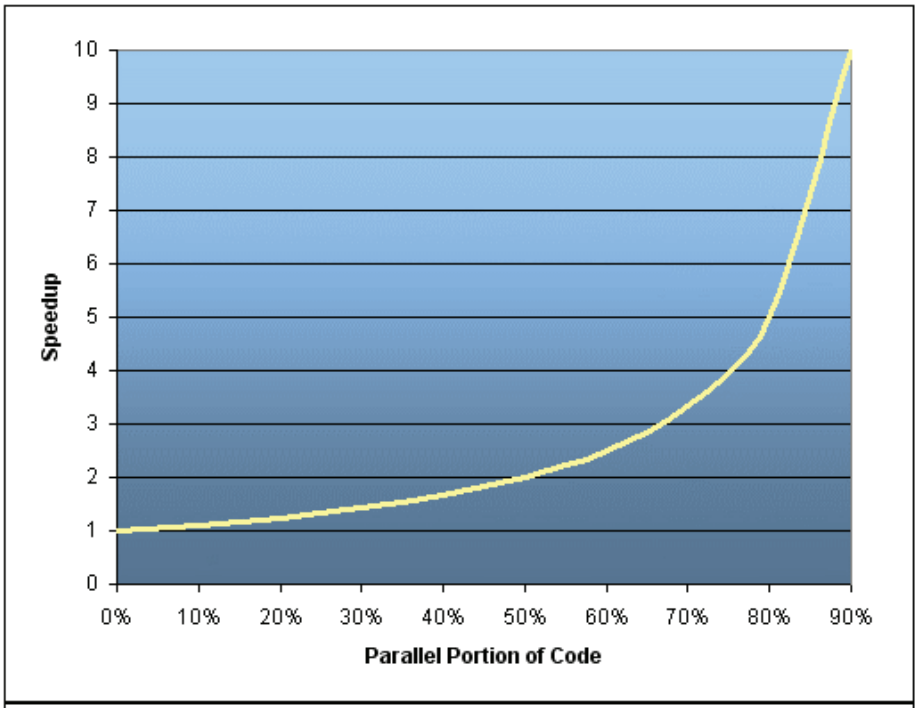


Ilustración 19 Incremento de la ganancia frente al porcentaje de ejecución paralela

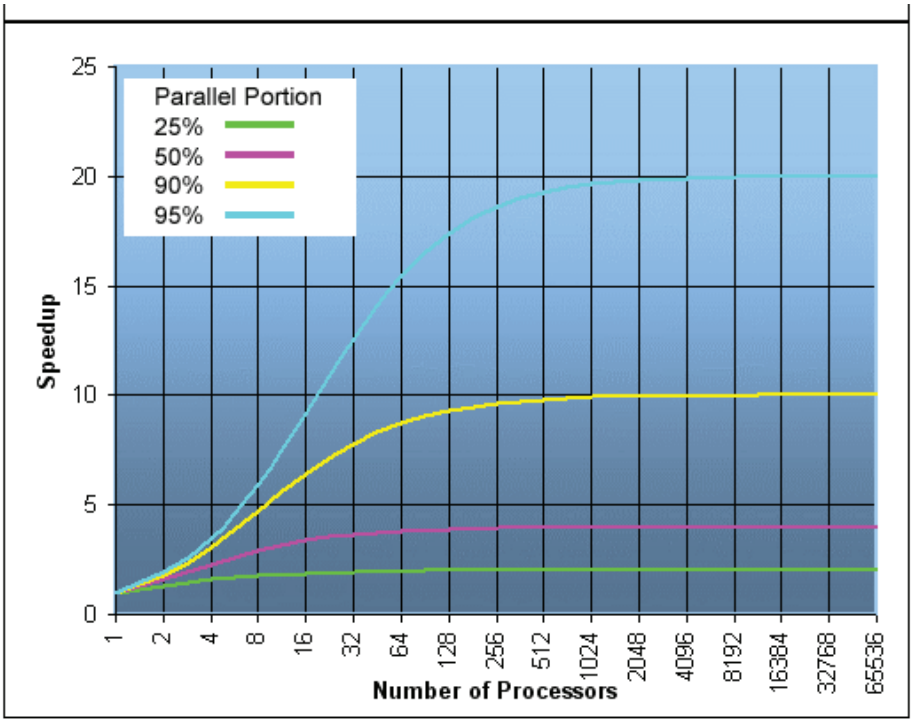


Ilustración 20 Incremento de la ganancia frente al número de procesadores y porcentaje de ejecución paralela

Como se ve en la Ilustración 19 el factor más importante para la reducción de tiempos de ejecución es el porcentaje de tiempo en regiones paralelas, creciendo de manera exponencial la ganancia. El número de procesadores es también muy importante pero como se ve en la Ilustración 20 se forma una función logarítmica limitando la ganancia que se puede obtener en un número relativamente bajo de procesadores si no se obtienen buenos porcentajes de ejecución en regiones paralelas. Es especialmente notable la diferencia entre el 90% de ejecución paralela (marcada en amarillo) y el 95% (marcada en azul), observándose que la ganancia se ve extremadamente afectada por el tiempo que se ejecuten estas regiones paralela y mucho menos por el número de procesadores que se utilizan.

2.3.5 Como paralelizar código

Para poder aprovechar al máximo las capacidades de paralelización y el porcentaje de tiempo que se ejecuta de forma paralela el código es importante entender como funcionan los modelos de ejecución paralela, especialmente el que emplea OpenMP ya que es la librería que se va a utilizar.

El modelo de programación que utiliza OpenMP es un modelo de paralelización mediante hilos, el cual es un modelo de programación de memoria compartida. En este modelo un proceso “pesado” puede tener múltiples procesos “ligeros” llamados hilos que permiten ejecutar fragmentos de código de manera simultánea.

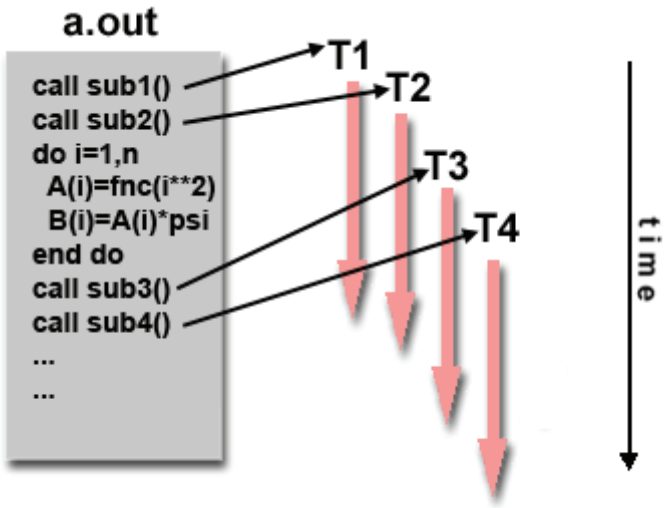


Ilustración 21 Ejecución simultánea mediante hilos [8]

En el ejemplo de la Ilustración 21 un proceso pesado ejecuta el programa *a.out*, dentro de este programa cada vez que se realiza una llamada a la función *call.subX()* se genera un nuevo hilo de ejecución Tx que se ejecuta paralelamente junto al resto del programa hasta que termina y devuelve su resultado al proceso pesado principal.

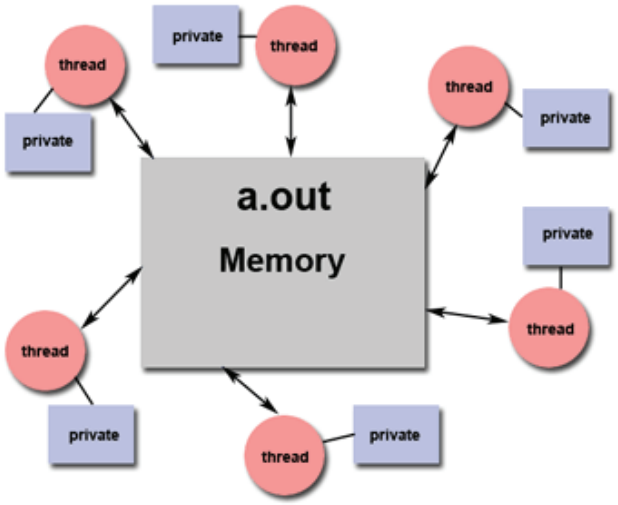


Ilustración 22 Distribución de la memoria del programa de ejemplo de la Ilustración 20

En la Ilustración 22 se muestra un diagrama de como se distribuye la memoria del programa *a.out.* Cada uno de los hilos se comunican con la memoria del programa de manera sincronizada y empleando mecanismos que eviten solapamientos de memoria o condiciones de carrera. A la vez se crean porciones privadas de memoria asociadas a cada hilo para que puedan disponer de sus propias variables para ejecutar los fragmentos de código que les corresponden.

Paralelizar programas ha sido desde siempre una actividad típicamente manual. El programador suele ser responsable de identificar el posible paralelismo e implementarlo. El desarrollo de estos códigos paralelos es muy costoso, ocupando un tiempo mayor de lo habitual para escribir bloques de código además de que es muy propenso a errores.

En la actualidad existen compiladores que paralelizan código de manera automática igual que con la vectorización, pero son mucho menos eficaces a la hora de reconocer áreas paralelizables y aplicar una paralelización eficiente y eficaz. Por lo tanto se va a utilizar la librería OpenMP, la cual mediante una serie de directivas permitirá indicar e nuestro compilador (en este caso GCC) que regiones se quieren paralelizar y actuar en consecuencia según el número de hilos que le indiquemos y que se estudiarán más adelante.

Para que la paralelización sea la mejor posible es esencial identificar dos secciones claves del código, los puntos calientes o *hotspots* y los cuellos de botella.

Los puntos calientes son las áreas del código que mayor coste computacional tienen y donde más tiempo se gasta de la ejecución del programa. Estas áreas son las que se deben identificar primero y actuar sobre ellas ya que es donde mayor ganancia de rendimiento se podrá obtener.

Los cuellos de botella son áreas del código no paralelizables. Estas áreas limitan mucho la velocidad de ejecución al ser desproporcionalmente lentas, y al no ser paralelizables no nos permiten obtener ninguna ganancia. Si estas áreas son excesivamente lentas y causan mucho retardo en las ejecuciones es conveniente estudiarlas para poder obtener mejores resultados. A veces es posible agilizar su ejecución o incluso hacer que sean paralelizables alterando el algoritmo que emplean o eliminando dependencias de datos en su ejecución. Las operaciones de entrada y salida son un ejemplo muy común de instrucciones no paralelizables que ralentizan la ejecución del código.

Una vez que se han identificado las secciones de código paralelizables lo primero que se debe hacer es dividir el problema en bloques más pequeños para realizar las acciones por separado. Este es un proceso muy similar al de la vectorización ya que en el fondo son dos procesos muy parecidos.

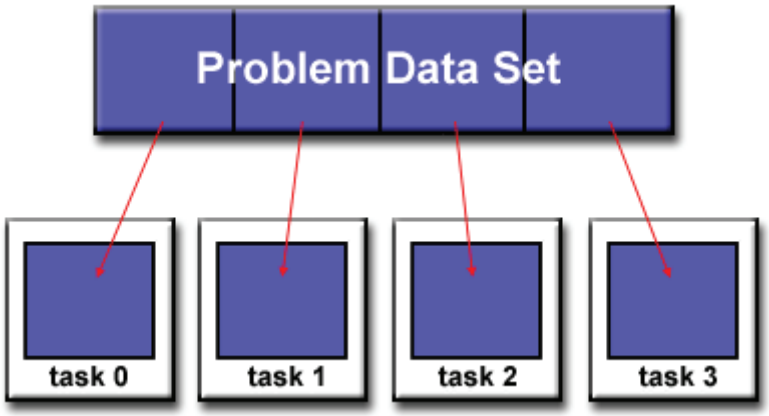


Ilustración 23 Partición del problema en bloques

En la Ilustración 23 se muestra una división típica de un conjunto de datos en cuatro bloques para ser ejecutados de manera paralela por cuatro hilos distintos.

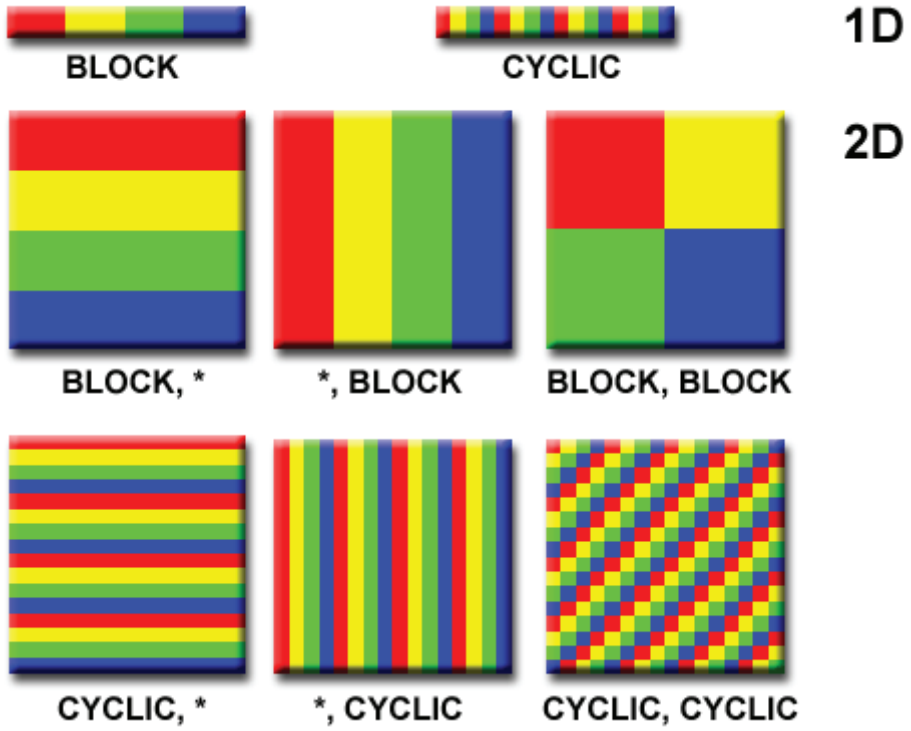


Ilustración 24 Organización de datos en memoria

Al igual que en la vectorización, en la paralelización también se puede ganar rendimiento si se organizan los datos de manera que estén consecutivos en la memoria de cada uno de los procesos que se van a ejecutar de manera paralela. En la Ilustración 24 se muestran varios ejemplos de organización del código de manera que se puedan acceder más rápidamente por los distintos hilos. En estos ejemplos los distintos colores representan bloques de información que deben interactuar en diversas operaciones y en caso de ser agrupados y asignados a los distintos hilos permiten agilizar mucho la ejecución. Además, al estar los datos de una sección paralelizable agrupados en memoria se puede ahorra trabajo a la sincronización que puede llegar a ser realmente costosa.

A diferencia de la vectorización que únicamente se podían dividir y organizar los datos para ganar rendimiento (descomposición del dominio), en la paralelización también se pueden dividir las tareas que va a realizar el programa (descomposición de funciones). El objetivo es dividir las tareas que va realizar el programa entre varios hilos (siempre y cuando se no tengan dependencias de las otras) en lugar de dividir una tarea entre varios hilos como se muestra en la Ilustración 25.

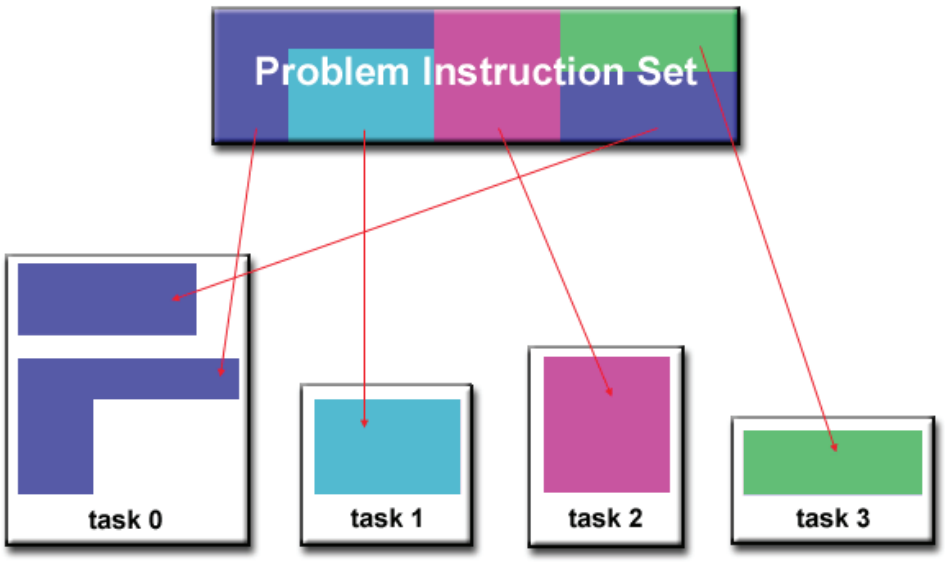


Ilustración 25 Descomposición funcional

Por último, se debe tener en cuenta la sincronización. Como ya se ha mencionado anteriormente pueden producirse condiciones de carrera y demás errores de sincronización que si no se tratan con cuidado podrían estropear el funcionamiento del código. Estas sincronizaciones típicamente se consiguen mediante semáforos, cerrojos o paso de mensajes entre los hilos indicando cuando pueden escribir y cuando deben bloquearse para mantener la consistencia en los resultados. El problema es que estos mecanismos suelen ser complicados de organizar, especialmente en un número elevado de hilos y si no se realizan bien pueden llegar a producir interbloqueos entre procesos dejándolos eternamente detenidos y muchos errores en los resultados.

Para solucionar estos problemas disponemos de OpenMP que realiza de manera automática la sincronización entre los hilos generados, ahorrando mucho tiempo de desarrollo y de corrección de errores. Esto último puede ser extremadamente costos, especialmente según aumenta el tamaño del código por lo que un sistema de sincronización automática con garantías (aunque no al 100%) de que no causará problemas es un ahorra extraordinario de esfuerzo para el programador.

2.3.5.1 Paralelización mediante OpenMP

La paralelización en OpenMP es muy parecida a la vectorización. Tiene el mismo funcionamiento mediante directivas y únicamente es necesario alterar ligeramente las que ya se han explicado en el apartado de vectorización.

La sintaxis de una directiva paralela es la siguiente:

```
#pragma omp parallel[clause[[,] clause]…]
        //Bucle for
```

Como ya se ha dicho, las clausulas son muy similares si no idénticas a las de la vectorización, pero hay algunas exclusivas para las directivas de paralelización.

Entre ellas se encuentran las clausulas de alcance de datos:

- Shared: Se comparte la variable entre todos los procesos.
- Private: Cada proceso tiene su propia copia de la variable.

A continuación se muestra un código de ejemplo de uso de estas cláusulas.

```
#pragma omp parallel shared(a,b,c,n) private(i)
for(i=0; i<n; i++){
      a[i] = b[i] + c[i];
}
```

El código anterior está indicando al compilador que genere una sección paralela en el bucle for y que las variables a, b, c y n sean compartidas mientras que la variable i es privada de manera que cada hilo pueda iterar libremente por el bucle pero no realice copias de a, b, c y n para ahorrar memoria.

Más cláusulas de alcance de datos:

- FIRSTPRIVATE: Las copias privadas de las variables son inicializadas con los valores originales de los objetos.
- LASTPRIVATE: Al terminar una región privada se le asigna a la variable el mismo valor que habría tenido si la ejecución fuese secuencial.

A continuación se muestra un código de ejemplo de uso de estas cláusulas.

```
#pragma omp parallel FIRSTPRIVATE(A) LASTPRIVATE(i)
for(i=0; i<n; i++){
      a[i] = A * b[i] + c[i];
}
```

De esta manera las copias privadas de A se inicializarán con el valor que tuviese A originalmente, y al terminar la ejecución el valor de la variable i será n en lugar de una fracción de n.

Para gestionar el uso del proceso principal y de lo hilos secundarios se emplea la directiva *master*.

```
#pragma omp parallel
{
      funcion();
      #pragma omp master
      { funcionMaestra();}
      #pragma barrier
      función();
}
```

La clausula *master* delimita una sección del código que únicamente será ejecutada por el proceso principal, impidiendo que el resto de hilos la ejecuten.

En el código anterior también se ha incluido la cláusula *barrier* (`#pragma barrier`). Esta clausula obliga a todos los hilos que la alcancen a esperar a que todos los demás hilos lleguen hasta ella. De esta manera se consigue que ninguno se adelante más de lo debido.

Además de las clausulas OpenMP ofrece una serie de funciones para aportar información al programador y configurar la cantidad de procesos que se quieren utilizar.

- `OMP_GET_NUM_THREADS()`: Devuelve el número actual de hilos en ejecución.
- `OMP_GET_THREAD_NUM()`: Devuelve el identificador del hilo.
- `OMP_SET_NUM_THREADS(n)`: Establece el número de hilos a utilizar.
- `OMP_GET_NUM_PROCS()`: Devuelve el número de procesadores de la máquina.
- `OMP_GET_MX_THREADS()`: Devuelve el número máximo de hilos que se utilizarán en la próxima región paralela.

A continuación se muestran varios ejemplos de códigos paralelizados y sus ganancias de tiempo. Todos estos ejemplos se han ejecutado con 6 procesadores por lo que la máxima ganancia teórica seria 6.

```
#pragma omp parallel for shared(a,b,c,MAX) private(i)
for(i=0; i<MAX; i++){
      a[i] = b[i] * c;
}
```

El código que se muestra arriba multiplica todos los elementos de un vector llamado *b* por otro elemento llamado *c* y los almacena en el vector *a.* Tras realizar 10000 veces la llamada a la función con un tamaño de vector de 100000 elementos se obtienen los siguientes tiempos de ejecución:

- Versión normal: 0.387021 segundos
- Versión paralelizada: 0.088861 segundos

Lo cual significa que la versión paralelizada es 4.355 veces más rápida, obteniéndose obtiene un 72.58% de la ganancia teórica. Aunque es un incremento muy grande está lejos del 100% de ganancia teórica. Esto se debe a que dentro de la función que se ejecuta existen otras operaciones no paralelizadas, la principal de ellas es la reserva de memoria para devolver un resultado. Esta operación podría hacerse externamente para ganar eficiencia pero se ha mantenido así ya que es la misma función que se va a emplear en la red neuronal.

Es importante indicar también que la indicación de variables *shared* o *private* en la directiva de paralelización no mejoran el rendimiento de manera apreciable.

NOTA: Cuando se compila el programa utilizando el compilador GCC es muy importante desactivar la vectorización automática que realiza el compilador, la cual se estudiará mas adelante. Para desactivarla basta con añadir las opciones `-O3 -fno-tree-vectorize`.

A continuación se muestra un código en el que se realiza el producto de dos vectores y los resultados se acumulan en una variable. Se emplea la clausula *reduction(+: var)* para indicar que es una suma acumulada del valor

```
#pragma mp parallel for reduction (+:a)
for(i=0; i<MAX; i++){
      a += b[i] * c[i];
}
```

El resultado de la operación es el mismo en la ejecución secuencial y en la ejecución paralela, lo cual es esencial para que las pruebas sean válidas.

Aplicando las mismas repeticiones y tamaño de vector que en el ejemplo anterior se obtienen los siguientes resultados:

- Versión normal: 0.000094 segundos
- Versión paralelizada: 0.000039 segundos

Lo cual significa que la versión paralelizada es 2.400 veces más rápida, obteniéndose obtiene un 40% de la ganancia teórica.

El siguiente código realiza la suma de dos vectores y almacena el resultado en un tercer vector.

```
#pragma omp parallel for shared(a,b,c,MAX) private(i)
for(i=0; i<MAX; i++){
      a[i] = b[i] + c;
}
```

Aplicando las mismas repeticiones y tamaño de vector que en el ejemplo anterior se obtienen los siguientes resultados:

- Versión normal: 0.000221 segundos
- Versión paralelizada: 0.000062 segundos

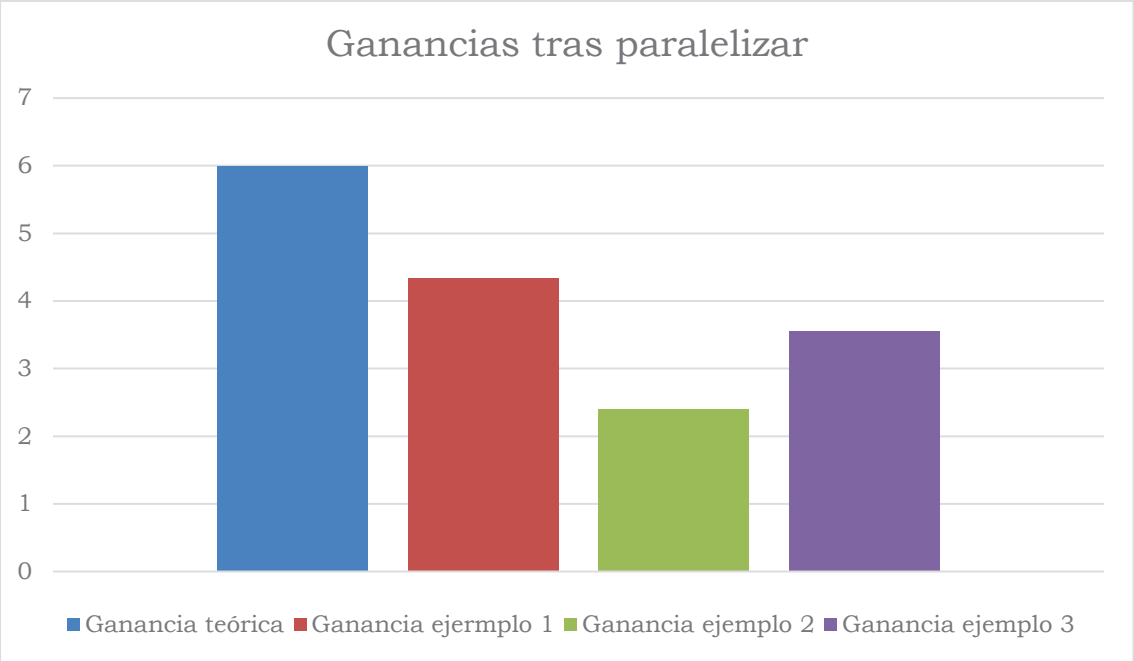
Lo cual significa que la versión paralelizada es 3.5491 veces más rápida, obteniéndose obtiene un 59.15% de la ganancia teórica.

Por último se puede producir una situación especial en la que se pretenda paralelizar un bucle dentro de otro bucle. Para esto es necesario utilizar la clausula `collapse(n)`. Esta clausula indicará al compilador que se trata de un bucle anidado de orden *n* de manera que pueda organizar los distintos hilos para realizar una ejecución paralela ordenada.

```
#pragma omp parallel for collapse(n)
```

2.3.6 Resultados y conclusiones

La paralelización puede aportar una gran ganancia de rendimiento si se hace de manera adecuada. Es esencial la sincronización entre los distintos hilos que se crean, ya que sin esta se pueden producir una gran cantidad de errores. Gracias a la librería OpenMP se puede paralelizar de una manera eficiente y mucho más sencilla que en los estilos de programación manuales.



Como se puede observar en el gráfico, las ganancias, aunque lejos de ser ideales, suponen un gran incremento en la velocidad de ejecución del código y si se aplican de manera eficaz puede reflejarse en gran medida en la red neuronal que se pretende paralelizar y vectorizar.

A pesar de que estos ejemplos que se han tratado son muy sencillos y podrían alcanzar ganancias muy próximas a la ganancia teórica esto no ocurre. La razón es que existen diversos cuellos de botella a la hora de ejecutar las operaciones. La principal es la reserva de memoria mediante la llamada a funciones `malloc()`. Estas reservas se realizan dentro de las funciones que hacen las operaciones entre vectores debido a que la orientación de la traducción de la red neuronal pretende que sea lo más parecida posible a la original ya existente. Esto causa una bajada en el rendimiento al generar un cuello de botella, pero sirve como ejemplo muy claro de los cuellos de botella que se pueden generar a la hora de programar, y que en un código con más complejo y con mayor cantidad de líneas de código existirán muchos de estos cuellos de botella.

2.4 Paralelización y vectorización combinadas

Hasta el momento se ha estudiado la paralelización y vectorización de código como dos técnicas independientes. En esta sección se va a analizar como combinar estas dos técnicas para obtener los mejores resultados posibles.

Para ello se va a tratar la paralelización únicamente mediante OpenMP ya que se han obtenido mejores resultados que en la vectorización mediante OpenMP, mientras que la vectorización se tratará con las dos opciones estudiadas en apartados anteriores, vectorización mediante OpenMP y vectorización automática mediante el compilador GCC.

2.4.1 Paralelización y vectorización con OpenMP

Es posible paralelizar y vectorizar bucles de manera simultánea empleando OpenMP. Para ello se combinan las cláusulas `#pragma omp parallel for` y `#pragma opm simd` creando una nueva cláusula, `#pragma omp parallel for simd`

Esta nueva cláusula se va a aplicar a los ejemplos mostrados en la sección de paralelización para comparar cuanto aumenta la ganancia al combinar ambas técnicas frente a usar únicamente la paralelización. No se compara frente a la vectorización aislada ya que su ganancia es mucho menor.

```
#pragma omp parallel for simd
for(i=0; i<MAX; i++){
      a[i] = b[i] * c;
}
```

El código que se muestra arriba multiplica todos los elementos de un vector llamado *b* por otro elemento llamado *c* y los almacena en el vector *a.* Tras realizar 10000 veces la llamada a la función con un tamaño de vector de 100000 elementos se obtienen los siguientes tiempos de ejecución:

- Versión paralelizada: 0.114505 segundos
- Versión paralelizada y vectorizada: 0.099804 segundos

Esto supone una ganancia de 1.147 en la versión combinada frente a la que únicamente paraleliza. Este resultado cuadra con los obtenidos en las secciones de vectorización mediante OpenMP.

A continuación se muestra un código en el que se realiza el producto de dos vectores y los resultados se acumulan en una variable. Se emplea la clausula *reduction(+: var)* para indicar que es una suma acumulada del valor

```
#pragma mp parallel for simd reduction (+:a)
for(i=0; i<MAX; i++){
      a += b[i] * c[i];
}
```

El resultado de la operación es el mismo en la ejecución secuencial y en la ejecución paralela, lo cual es esencial para que las pruebas sean válidas.

Aplicando las mismas repeticiones y tamaño de vector que en el ejemplo anterior se obtienen los siguientes resultados:

- Versión normal: 0.000062 segundos
- Versión paralelizada: 0.000040 segundos

Esto supone una ganancia de 1.569 en la versión combinada frente a la que únicamente paraleliza. Es una mejora mayor que en el caso anterior, probablemente debido a la operación de suma acumulada la cual permite agilizarse al usar vectorización ya que se divide la operación en bloques que finalmente se combinan.

El siguiente código realiza la suma de dos vectores y almacena el resultado en un tercer vector.

```
#pragma omp parallel for simd
for(i=0; i<MAX; i++){
      a[i] = b[i] + c;
}
```

Aplicando las mismas repeticiones y tamaño de vector que en el ejemplo anterior se obtienen los siguientes resultados:

- Versión normal: 0.000036 segundos
- Versión paralelizada: 0.000029 segundos

Esto supone una ganancia de 1.227 en la versión combinada frente a la que únicamente paraleliza.

2.4.2 Paralelización y vectorización mediante OpneMP y GCC

En esta sección se va a estudiar la combinación de paralelización mediante OpenMP y vectorización automática usando el flag -O3 de GCC que habilitará la máxima optimización posible además de activar la vectorización automática.

A continuación se muestran los mismos ejemplos que en secciones anteriores.

```
#pragma omp parallel for
for(i=0; i<MAX; i++){
      a[i] = b[i] * c;
}
```

Los resultados obtenidos son:

- Versión paralelizada y vectorizada mediante OpenMP: 0.099804 segundos
- Versión paralelizada y vectorizada mediante GCC: 0.068612 segundos

Se obtiene una ganancia de 1.455 en la versión vectorizada automáticamente frente a la manual.

```
#pragma mp parallel for
for(i=0; i<MAX; i++){
      a += b[i] * c[i];
}
```

Los resultados obtenidos son:

- Versión paralelizada y vectorizada mediante OpenMP: 0. 000040 segundos
- Versión paralelizada y vectorizada mediante GCC: 0. 000040 segundos

Se obtiene una ganancia de 1 en la versión vectorizada automáticamente frente a la manual. Esto significa que no ha habido ninguna reducción en los tiempos de ejecución. Por lo tanto la mayor mejora es la conseguida en la versión paralelizada y vectorizada usando OpenMP, la cual ha producido una ganancia de 1.569 frente a la paralelización normal.

Tras compilar el programa con el flag -fopt-info-vec-missed se comprueba que el compilador no ha sido capaz de vectorizar automáticamente el bucle a causa de la suma acumulada que se realiza, por lo tanto, el resultado es igual en la versión paralelizada como en la paralelizada y vectorizada automáticamente mediante GCC.

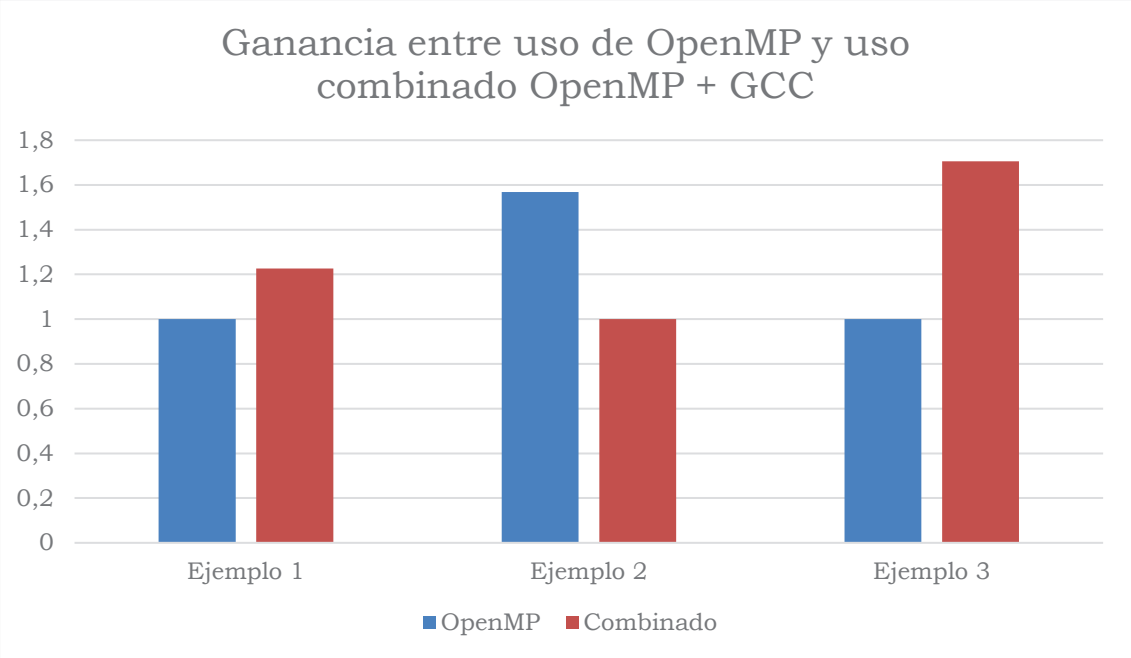
```
#pragma omp parallel for simd
for(i=0; i<MAX; i++){
      a[i] = b[i] + c;
}
```

Los resultados obtenidos son:

- Versión paralelizada y vectorizada medinate OpenMP: 0. 000029 segundos
- Versión paralelizada y vectorizada mediante GCC: 0.000017 segundos

Se obtiene una ganancia de 1.706 en la versión vectorizada automáticamente frente a la manual.

2.4.3 Resultados y conclusiones



Como se puede observar, los resultados mediante la combinación de paralelización mediante OpenMP y vectorización mediante GCC tienden a ser mejores pero únicamente sirven en ejemplos sencillos como son el 1 y el 3. En estos ejemplos el bucle realizaba un producto entre un vector y un elemento y una suma entre dos vectores respectivamente. Al no tener situaciones especiales el compilador ha sido capaz de vectorizar automáticamente sin ningún problema. Sin embargo en el ejemplo 2 en el que se emplea una suma

acumulada el compilador no ha sido capaz de vectorizar automáticamente los bucles ya que no ha sabido como tratar la variable sobre la que se acumulaba.

Con esta información se puede concluir que como primera acción es más rápido y eficaz usar el método combinado en el que la vectorización la trate el compilador, pero si se pretende alcanzar un alto nivel de ganancia se deberá investigar que fragmentos de código no ha sido capaz de vectorizar el compilador y como actuar sobre ellos, empleando las cláusulas descritas en los apartados anteriores para tratar de conseguir el máximo rendimiento posible sin comprometer los resultados obtenidos.

2.5 Traducción de la red neuronal

Como se ha explicado en la introducción, se dispone de una red neuronal escrita en Python. Para poder entender mejor el funcionamiento y tener más libertad para controlarla y utilizar la librería OpenMP se va a traducir dicha red neuronal al lenguaje de programación C.

Esto supondrá un desafío ya que Python es un lenguaje mucho más sencillo que C, permitiendo al programador mucha libertad a la hora de manejar distintos tipos y listas sin tener que realizar reservas de memoria. Además el extenso uso de la librería *numpy* facilita mucho el desarrollo de cálculos aritméticos en Python pero obliga a realizar versiones en C de todas las funciones empleadas de la librería.

2.5.1 Diferencias entre Python y C

Dentro de los lenguajes de alto nivel, C se considera como el de más bajo nivel o de los más bajos. Esto significa que está más próximo al nivel de instrucción del ordenador. Típicamente esto hace que se considere que es un lenguaje más "difícil" que otros como Python ya que el programador debe manejar mucho más el ordenador y requiere un entendimiento más profundo de lo que se está haciendo, provocando que cada vez sea menos común su uso. Pero si se maneja bien esto permite mucha más libertad y precisión a la hora de desarrollar programas y por eso es por lo que es el lenguaje elegido para este proyecto.

Por el contrario Python es un lenguaje de más alto nivel. La gestión de memoria se realiza de manera automática, lo cual ahorra esfuerzo al programador, y su sintaxis es más simple. Además su gran popularidad ha provocado que haya muchísimas librerías disponibles, agilizando mucho el desarrollo de nuevos programas y aumentando la comodidad para los usuarios. Por ello se ha vuelto uno de los lenguajes más populares del mundo, ya que su manejo es sencillo, permite una introducción rápida y fácil a la programación y al ser tan popular se dispone de muchísima documentación y apoyo online.

En lo que respecta al rendimiento, C es mucho más rápido y eficiente que Python si se maneja de manera adecuada. C es un lenguaje compilado mientras que Python es interpretado. Esto significa que C realiza un análisis previo del código escrito reservando la memoria necesaria y generando las instrucciones de código máquina que se van a ejecutar. Por el contrario Python al ser interpretado realiza todo este proceso a la vez que se va ejecutando el código por lo que con cada ejecución también se realiza la "compilación". La principal desventaja de esto es que aumenta en gran medida los tiempos de ejecución, pero a cambio aporta muchas facilidades al programador para escribir código. Al interpretarse el código según se ejecuta no es necesario declarar tipos de variables ya que se evaluarán según vayan apareciendo y se reservará memoria acorde al tipo, esto además permite reutilizar nombres de variables con distintos tipos ya que se reasignan según se encuentran. Tampoco es necesario reservar memoria al realizarse esta tarea de manera automática, pero se aumenta considerablemente la eficiencia ya que la reserva de memoria automática es una tarea muy costosa.

2.5.2 Estructura de la red neuronal

La red neuronal se ha diseñado como una clase compuesto de 6 funciones principales y 4 estructuras de datos, siendo las funciones las siguientes:

- Feedforward: Recibe como input un vector con los valores necesarios para calcular los resultados que devuelve la función neuronal (función sigmoid).
- SGD: Es la función principal de la red neuronal. Recibe como parámetros los conjuntos de datos de entrenamiento y testeo, además de el tamaño en que se van a subdividir estos conjuntos y el número de iteraciones de entrenamiento que se quieren ejecutar.
- Update_mini_batch: Actualiza los pesos y biases de la red neuronal usando un subconjunto (mini batch) de datos aplicando el algoritmo de *backpropagation* y descenso gradiante según un ritmo de aprendizaje pasado como parámetro.
- Backprop: Recibe un vector de pesos y un vector de resultados y devuelve una tubla de pesos y biases con el gradiante calculado usando los inputs.
- Evaluate: evalúa la tasa de acierto de la red neuronal usando le conjunto de testeo.
- Cost_derivative: Devuelve el resultado de las derivadas parciales necesarias para el cálculo del coste gradiante.

Y las estructuras de datos las siguientes:

- Num_layers: Numero de capas de la red neuronal.
- Sizes: Tamaño de cada una de las capas de la red.
- Biases: Vectores de biases de cada capa.
- Weights: Vectores de pesos de cada capa.

Por último hay dos funciones adicionales externas a la clase:

- Sigmoid: Calcula la función de activación sigmoide de un conjunto de neuronas.
- Sigmoid_prime: Calcula una derivación de la función sigmoide de un conjunto de neuronas.

2.5.3 Proceso de traducción y dificultades

El primer problema que ha supuesto la traducción ha sido lo escueto que es el código en Python. Al no declarar tipos de variables ni tamaños de vectores se debía averiguar uno por uno el contenido de cada estructura de datos y esto ha supuesto un enorme consumo de tiempo.

Respecto a la programación en primera instancia el principal problema ha sido la gestión de memoria y reserva de espacios. Esta tarea, como ya se ha mencionado previamente, la realiza el compilador de Python de manera automática mientras que en C se debe hacer de forma manual. El ejemplo mas claro, y uno de los primeros que se deben tratar, es el de los vectores de pesos.

Supongamos que la red neuronal se compone de tres capas. La primera capa (entrada) de entrada dispone de 784 neuronas, la segunda capa (intermedia) dispone de 30 neuronas y la última capa (salida) dispone de 10. Esto creará dos bloques de pesos y dos bloques de biases. El primer conectará la primera capa con la segunda y el segundo conectará la segunda capa con la tercera.

El primer bloque se forma por una matriz de 784*30 de manera que cada neurona de la capa de entrada tiene un peso asociado con cada neurona de la capa intermedia. Y de 30*10 en el segundo bloque por la misma razón.

En la versión de Python esta estructura se genera como una matriz bidimensional, en la traducción a C se ha estructurado como un único array consecutivo de datos para agilizar los accesos a memoria como se explica en el

apartado de vectorización de este documento y se visualiza en las Ilustraciones 13 y 14.

La dificultad de manejar esta estructura de datos como un vector consecutivo con todos los elementos es que hace mucho más difícil el acceso, pero para solucionar este problema se han creado dos funciones auxiliares:

- `double* getWeights(int n)`
- `double * getBiases(int n)`

La tarea de estas funciones es devolver los vectores de pesos y biases según la capa que se le indique como input el resultado es un puntero a la posición de memoria en la que comienza el vector deseado. Es importante saber el tamaño con el que se está trabajando y manejar con mucha delicadeza los datos ya que si no se obtienen correctamente se puede estropear toda la estructura.

El siguiente problema y uno de los que más problemas puede llegar a causar si no se está atento es la gestión de memoria.

Típicamente los lenguajes modernos realizan las reservas de memoria sin que el programador deba encargarse de ellas pero también la liberan, lo cual es esencial si no se quiere saturar la máquina en la que se ejecuta.

En C la reserva se realiza con la función `malloc()` o versiones más adecuadas para la vectorización como ya se ha explicado en apartados anteriores, y la liberación se realiza con la función `free()`. Es extremadamente importante liberar la memoria en cuanto ya no es necesario emplear su contenido, ya que si no se hace de forma manual se puede saturar la máquina y ralentizar extremadamente la ejecución.

2.5.3.1 Funciones de Numpy

Numpy es una librería de Python para cálculos matemáticos y aritméticos enormemente extendida, hasta el punto de que se usa casi por defecto en los programas escritos en Python. Es especialmente útil por sus funciones de operaciones entre vectores, las cuales se emplean enormemente en redes neuronales.

Por desgracia C no dispone de una librería así, por lo que todas estas funciones se deberán programar manualmente causando una ralentización en el desarrollo, pero aumentando enormemente la comprensión de lo que se está haciendo y permitiendo muchísima libertad para optimizar estas operaciones.

Las principales funciones de Numpy que se han replicado son:

- Producto de todos los elementos de un vector por un único elemento.
- Producto de dos vectores y resultado acumulado en un valor.
- Producto de una matriz por un vector.
- Suma de dos vectores.
- Resta de dos vectores.
- Producto de dos vectores.
- Máximo valor de un vector.

Varias de estas funciones se han evaluado en los apartados de paralelización y vectorización para calcular la ganancia obtenida tras aplicar la librería OpenMP con resultados muy positivos por lo que a la hora de aplicar la paralelización y vectorización a la red neuronal estas serán las principales funciones sobre las que se hará por ser candidatas para aportar un gran aumento de rendimiento y porque son funciones que ocupan una gran parte del tiempo de ejecución total.

2.5.4 Conclusiones

La traducción de la red neuronal ha sido mucho más trabajosa de lo que inicialmente se esperaba. Al no declararse ni tipos ni tamaños en los códigos de Python ha resultado tremendamente costos averiguar que correspondía a cada uno. Esta es una desventaja de Python, resulta muy legible entender lo que hace el código de manera superficial, pero si se pretende profundizar en su funcionamiento requiere mucho más esfuerzo que en otros lenguajes tipados.

La gestión de memoria para realizar reservas y liberaciones ha resultado bastante costosa hasta el punto de que ha tomado por lo menos un tercio del desarrollo entre realizarla y corregir errores producidos. Se entiende por qué cada vez menos lenguajes la realizan de manera manual, especialmente con las capacidades de máquinas modernas en las que los espacios de memoria no son tan ajustados como para requerir niveles extremos de optimización.

La traducción de las funciones de Numpy inicialmente parecía que iba a ser un consumo de tiempo adicional que no aportaría demasiado al desarrollo del proyecto pero ha resultado ser clave para su funcionamiento. La posibilidad de acceder de manera interna a estas funciones al haberse creado versiones propias permite unos niveles de optimización muy superiores, y sobre todo permite aplicar paralelización y vectorización de una manera mucho más efectiva, además de separarlas para realizar un análisis aislado mediante el compilador GCC y poder visualizar los resultados de la vectorización automática.

Como conclusión resulta totalmente comprensible que no se emplee un lenguaje como C para la realización de este tipo de proyectos, especialmente si se aplica a las industrias de desarrollo de software en las que la velocidad de desarrollo y legibilidad del código son métricas extremadamente importantes. Pero si se decide hacer en un lenguaje como C se abre un mundo de posibilidades para mejorar el rendimiento y permite profundizar mucho más en el funcionamiento del código que se está desarrollando.

2.6 Paralelización y vectorización de la red neuronal

En este apartado se va a paralelizar y vectorizar la red neuronal aplicando las técnicas aprendidas en los apartados anteriores y evaluar los resultados conseguidos.

2.6.1 Paralelización de la red neuronal

Para poder aplicar el conocimiento adquirido en apartados anteriores primero se deberá identificar cuales son las secciones del código que más tiempo consumen y si es posible paralelizar y vectorizar dichas secciones.

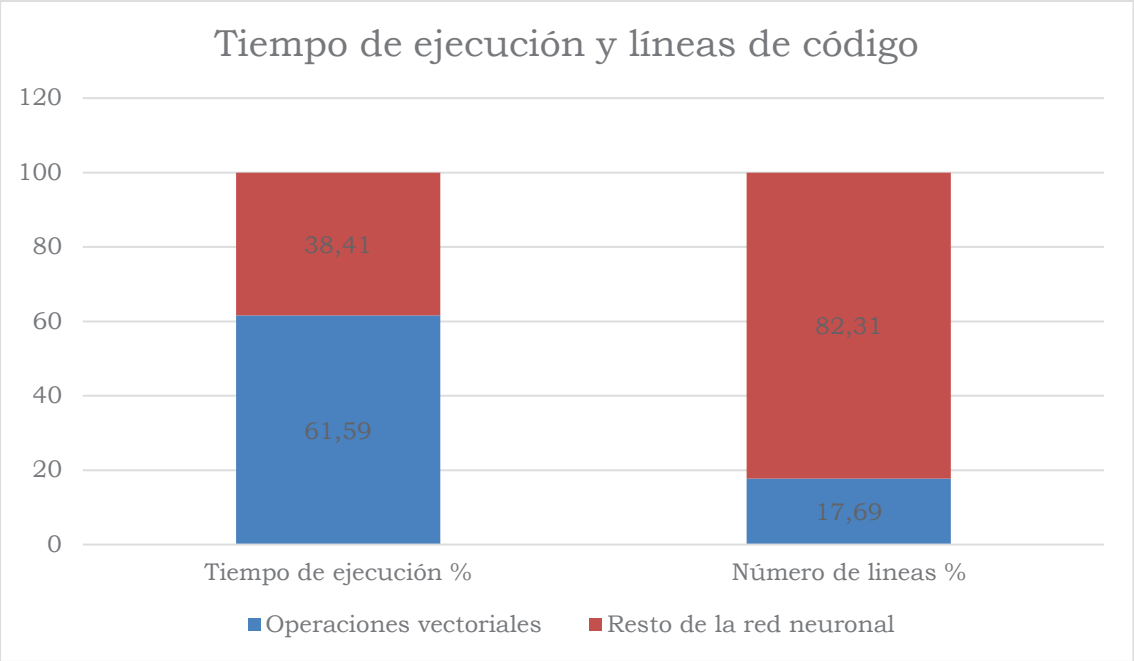
Las funciones candidatas a ser paralelizadas y vectorizadas son, como se ha mencionado en apartados anteriores, aquellas que realizan los cálculos de la red neuronal entre vectores. En concreto en este caso son las funciones que se han traducido de la librería Numpy de Python a una versión propia en C. Para saber que porcentaje del tiempo de ejecución ocupan se va a emplear la función de OpenMP llamada omp_get_wtime() la cual almacena en un valor *double* la hora en la que se llama a esta función. Realizando una llamada al comienzo de la función y otra al final y restando estos valores se obtiene el tiempo de ejecución en segundos. Realizando esta operación en todas las funciones traducidas y acumulando el valor en una variable se obtendrá el valor total que consumen las funciones de cálculo vectorial respecto a la ejecución.

Para agilizar la obtención de resultados se han realizado los cálculos con un número reducido de muestras, 600 imágenes de entrenamiento y 100 de evaluación. De esta forma se podrán realizar más repeticiones de las pruebas para obtener valores medios más fiables en lugar de pocas ejecuciones de gran tamaño.

A continuación se muestran los resultados obtenidos:

- Tiempo medio de ejecución de 20 repeticiones = 3.150399 segundos
- Tiempo medio de ejecución de funciones vectoriales = 1.940495 segundos

Esto supone que un 61.59% del tiempo total de ejecución de la red neuronal se emplea en las funciones de cálculo vectorial. Teniendo en cuenta que las funciones de cálculo vectorial representan 115 líneas de código y el resto de la red neuronal se compone de 650 líneas queda claro que va a resultar mucho más fácil actuar en primera instancia sobre estas funciones para obtener mejores resultados.



El primer paso que se va a realizar va a ser el de la vectorización automática usando el compilador GCC con el *flag -O3 -fopt-info-vec-optimized* activado para comprobar en que casos se han vectorizado los bucles de manera automática y en caso de que no se hayan vectorizado se aplicarán directivas SIMD como las estudiadas en el apartado de vectorización para guiar al compilador de manera que realice la vectorización.

Tras la compilación y la evaluación del log de vectorización se comprueba que todas las funciones de cálculo vectorial se han vectorizado correctamente salvo dos.

Estas dos funciones son casos especiales que se van a tratar mediante clausulas SIMD estudiadas en el aparatado de vectorización.

La primera función no vectorizada realiza el producto de los elementos de dos vectores y acumula el resultado en único valor que va sumando como se muestra a continuación:

```
for(i=0; i<MAX; i++){
      result += v[i] * u[i];
}
```

Siguiendo lo estudiado en el apartado de vectorización se deberá escribir una cláusula que indique al compilador que la variable suma es una reducción sobre la que van a colapsar los resultados de las operaciones vectoriales mediante una operación de suma, lo cual se hace con la siguiente clausula:

```
#pragma omp simd reduction(+:result)
```

Tras compilar de nuevo el programa y revisar el log se puede ver que esta vez el bucle si que ha sido vectorizado.

```
operaciones.h:34:20: note: loop vectorized
```

Ilustración 2611 Resultado de vectorización automática

El siguiente caso no vectorizado se trata de otra situación especial tratada en el apartado de vectorización. En este caso no se ha vectorizado automáticamente el bucle debido a que se trata de dos bucles anidados:

```
for(i=0; i<repeticiones; i++){
	aumento = i*tam_vector_u;
	for(j=0; j<tam_vector_u; j++){
		auxiliar[j] = v[+j+aumento];
	}
}
```

Lo primero que se puede plantear para solucionar el problema sería emplear la clausula *colapse(2),* la cual indica al compilador que se trata de un bucle anidado de orden 2 para que organice la vectorización de manera que distribuya los recursos entre los dos bucles. Pero si se observa más detenidamente se comprueba que existe una dependencia de datos producida en el primer bucle en la operación siguiente:

```
aumento = i*tam_vector_u;
```

Esta operación rellena la variable *aumento* que luego es empleada en el siguiente bucle para realizar otro cálculo de manera que se genera una dependencia de datos que impide una correcta vectorización, como se puede observar en el log de la vectorización:

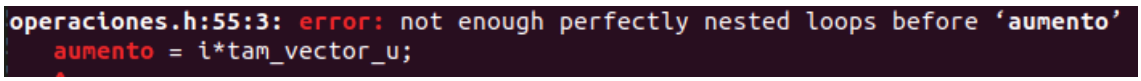


Ilustración 27 Error de dependencia de datos

Si no se pueden vectorizar ambos bucles, o por lo menos el primero resulta inviable vectorizar el segundo ya que en su interior no se realizan operaciones aritméticas candidatas a vectorización si no copia y asignación de información, la cual no es vectorizable pero sí paralelizable.

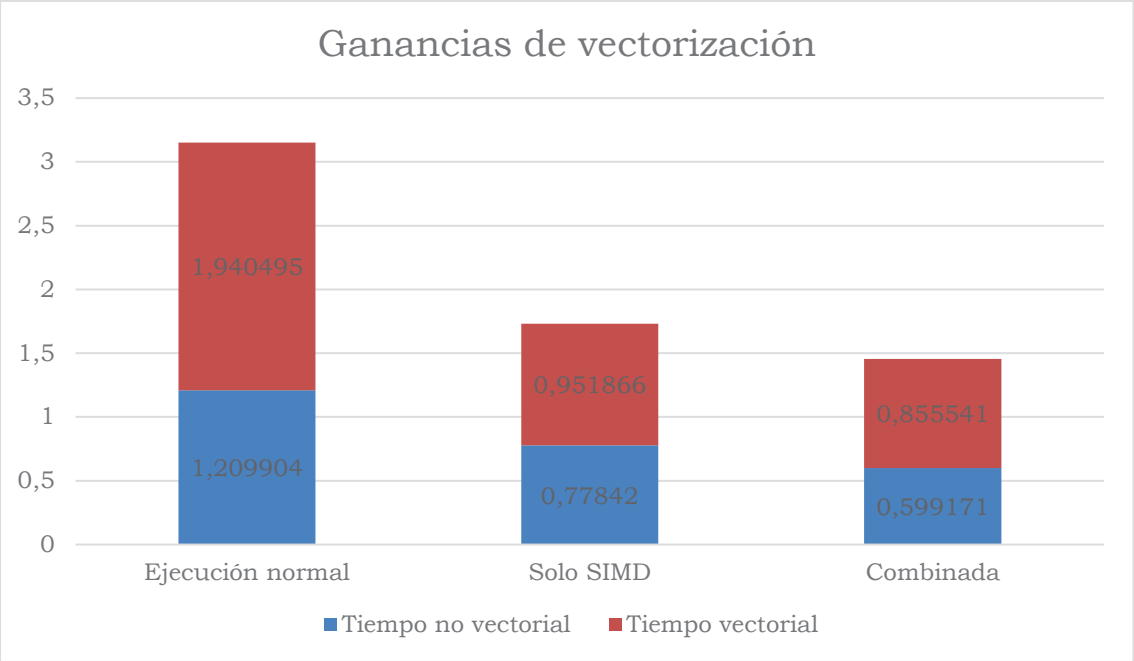
Para comprobar los resultados de la vectorización automática junto a la optimización máxima del compilador se ha compilado y ejecutado el código de dos maneras diferente. Con la opción -O3 pura que optimiza y vectoriza al máximo posible y con la opción -O3 incluyendo el flag -fno-tre-vectorize de manera que no vectorice el código y se pueda comparar.

Mediante la ejecución sin vectorización automática empleando directivas SIMD indicando al compilador que no realice vectorización automática se obtienen los siguientes resultados:

- Tiempo de ejecución: 1.730286 segundos
- Tiempo en operaciones vectoriales: 0.951866 segundos

Mientras que empleando la ejecución combinada de directivas SIMD junto a la vectorización automática se obtienen los siguientes resultados:

- Tiempo de ejecución: 1.454712 segundos
- Tiempo en operaciones vectoriales: 0.855541 segundos



Como se ve en el gráfico anterior se obtiene un descenso considerable del tiempo de ejecución original, pasando de 3.150399 segundos originalmente a 1.454712 segundos aplicando la máxima optimización y vectorización. Esto representa un descenso de 1,695687 segundos en el tiempo de ejecución siendo un 53,82%. Por lo que la ganancia obtenida respecto a la ejecución normal es del 46,18%.

El tiempo de ejecución de las operaciones vectoriales ha descendido 1,084954 segundos, convirtiéndose en un 55,89% del valor original, lo cual significa que la ganancia en las operaciones vectoriales es del 44,11%.

Es importante notar que una parte de estas mejoras de tiempo en la ejecución de la red neuronal vienen debido a la optimización que aplica GCC gracias al *flag -O3*. Además de que muchas operaciones dentro de la propia red neuronal han sido vectorizadas de manera automática por el compilador sin ser las indicadas mediante directivas SIMD o estar incluidas en las funciones de cálculo vectorial indicadas previamente.

2.6.2 Vectorización de la red neuronal

Para vectorizar la red neuronal se van a aplicar las directivas de OpenMP estudiadas en apartados anteriores. El objetivo es lograr una mejora en la ganancia sin perder los avances logrados mediante la vectorización, o identificar situaciones en las que se obtiene una mayor ganancia paralelizando en vez de vectorizando de manera que se obtengan los mejores resultados posibles.

Al igual que en el apartado anterior las funciones candidatas a ser vectorizadas son las que realizan operaciones entre vectores al ser las que ocupan la mayor parte del tiempo de ejecución y al estar compuestas de bucles relativamente simples son fáciles de paralelizar. Pero en este caso también se pueden tener en cuenta otros bucles que se realizan en el programa principal que, al no realizar operaciones aritméticas no han sido candidatos a ser vectorizados pero tienen potencial para ser paralelizados.

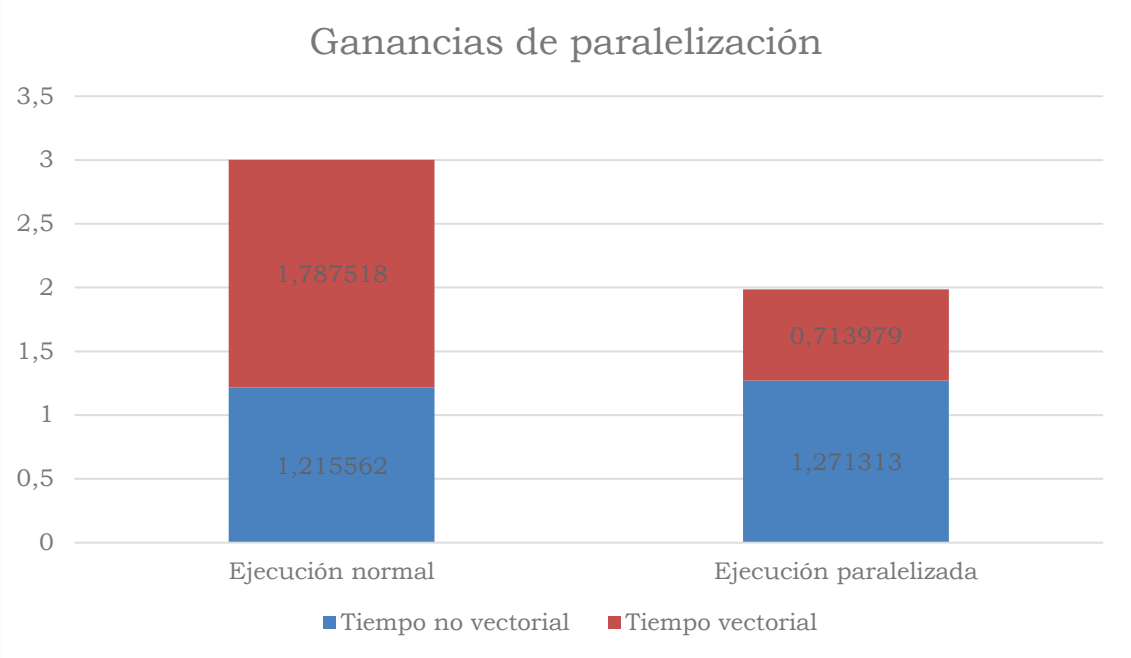
Es muy importante tener en cuenta que la paralelización es más costosa computacionalmente que la vectorización, es decir, las operaciones vectorizadas no requieren tanto esfuerzo computacional para formarse y ejecutarse mientras que las operaciones paralelizadas requieren la creación de múltiples threads y su sincronización por lo que será esencial aplicar paralelización únicamente en

las áreas del código que sean lo bastante grandes como para compensar el coste adicional de la creación y sincronización de varios hilos de ejecución.

Dado que la paralelización es mucho más susceptible a fallos se deberá estar mucho más atento a posibles errores.

La primera acción realizada es aplicar las directivas OpenMP a todas las operaciones de cálculo vectorial para ver que posibles ganancias se obtienen, pero el resultado inicial es malo, obteniéndose pérdida de rendimiento en la ejecución del código del orden de 2 segundos.

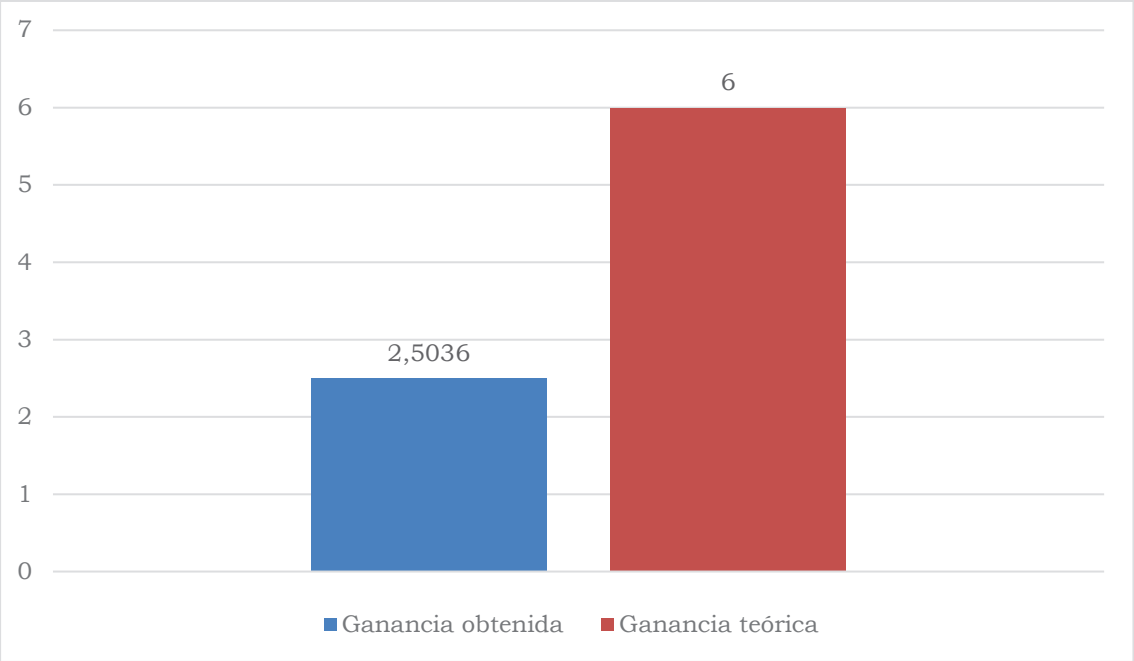
Tras una investigación rápida se puede observar que el problema proviene de una operación de cálculo con suma acumulada en la que se debe aplicar la cláusula *reduction*. En este caso la cláusula ha producido una ralentización en el código por lo que se ha optado por no paralelizar esta sección ya que las ganancias únicamente mediante vectorización son mejores.



En el gráfico anterior se muestra la ganancia de rendimiento de la red neuronal tras una ejecución paralelizando las operaciones vectoriales.

En el tiempo de ejecución de las secciones paralelizadas se aprecia un descenso al 39,94% del valor de una ejecución normal, lo cual supone una ganancia del 60,06%.

Aunque no son valores malos si se ven fuera de contexto si se tiene en cuenta que la ganancia ideal sobre esas secciones de código significaría descender el tiempo de ejecución a 0.297919 segundos frente a los 0.713979 que se han logrado se considera que son resultados malos.



La ganancia obtenida dista mucho de la ganancia real obtenida. Esto se debe a varios motivos.

Como se ha explicado previamente en los bucles cuyo resultado se acumula en una suma resulta más eficiente aplicar vectorización y por lo tanto la ganancia que se podría obtener de esa sección de código no se ha tenido en cuenta.

En muchos casos en los que los cálculos son en bucles de pocas iteraciones no resulta eficiente realizar paralelización ya que el coste de generar nuevos hilos y sincronizarlos es mucho más elevado que el de realizar la ejecución simplemente de manera secuencial y por eso se pueden llegar a apreciar diferencias muy grandes entre las pruebas realizadas en el apartado de paralelización de este documento a la aplicación de un caso real. En dichas pruebas se realizaba un número muy elevado de operaciones (del orden de 20000 repeticiones) sobre vectores de miles de elementos, sin embargo, en un caso real puede suceder que no se realicen más que una decena de operaciones sobre un vector de pocos cientos de elementos.

La sincronización causa una gran cantidad de errores y tratar de solucionarlos puede llegar a ralentizar la ejecución del programa en vez de agilizarla. Este es un error típico de los problemas de concurrencia y se planteaba como una posible solución el uso de la librería OpenMP pero no ha producido los resultados que se esperaban.

En los casos especiales que se han estudiado en el apartado de vectorización de la red neuronal se ha planteado la solución mediante paralelización sin demasiado éxito.

En el ejemplo ya mencionado de la suma acumulada se ha concluido que se obtiene una mayor ganancia al aplicar vectorización que al aplicar paralelización por lo que se ha planteado dejar así el código para no perder las ganancias obtenidas.

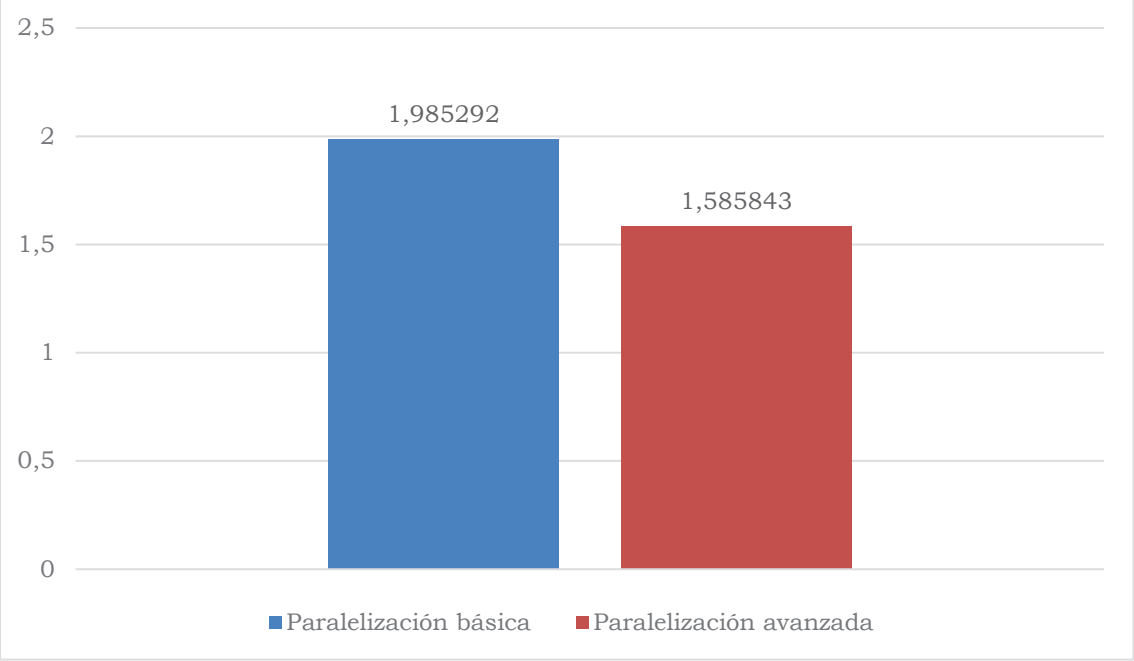
En el caso de bucles anidados se ha pretendido realizar la misma solución que con la vectorización, indicar mediante una cláusula *colapse(2)* que esa sección

del código se compone de dos bucles anidados pero el resultado ha sido el mismo. Al existir una dependencia de datos no esquivable en esa sección del código no se pueden paralelizar dichos bucles y por lo tanto se debe dejar la ejecución en forma secuencial.

Es importante notar que, aunque descienda el tiempo de ejecución de las zonas paralelizadas también aumenta ligeramente el de las zonas secuenciales. Esto se debe a que las reiteradas llamadas mediante bucles a funciones de cálculo vectorial hacen que estas deban realizar muchas tareas de creación de hilos y sincronización, y aunque la mayor parte de este tiempo está contabilizado dentro de la propia paralelización hay algunas secciones que quedan fuera y provocan una ralentización en la ejecución secuencial, aunque sea poco apreciable.

Como los valores han sido poco satisfactorios y la paralelización permite más libertad de acción que la vectorización se han paralelizado varias funciones adicionales que se ejecutaban en lo que hasta ahora se consideraba la parte secuencial.

Estas funciones son muy diversas pero los casos más notables son los de reserva e inicialización de memoria y de movimiento de datos. Estas funciones no eran candidatas a ser vectorizadas ya que la vectorización únicamente permite realizar operaciones aritméticas relativamente simples o con situaciones especiales muy concretas, mientras que la paralelización permite ampliar las posibilidades sobre las que actuar como en los ejemplos mencionados antes.



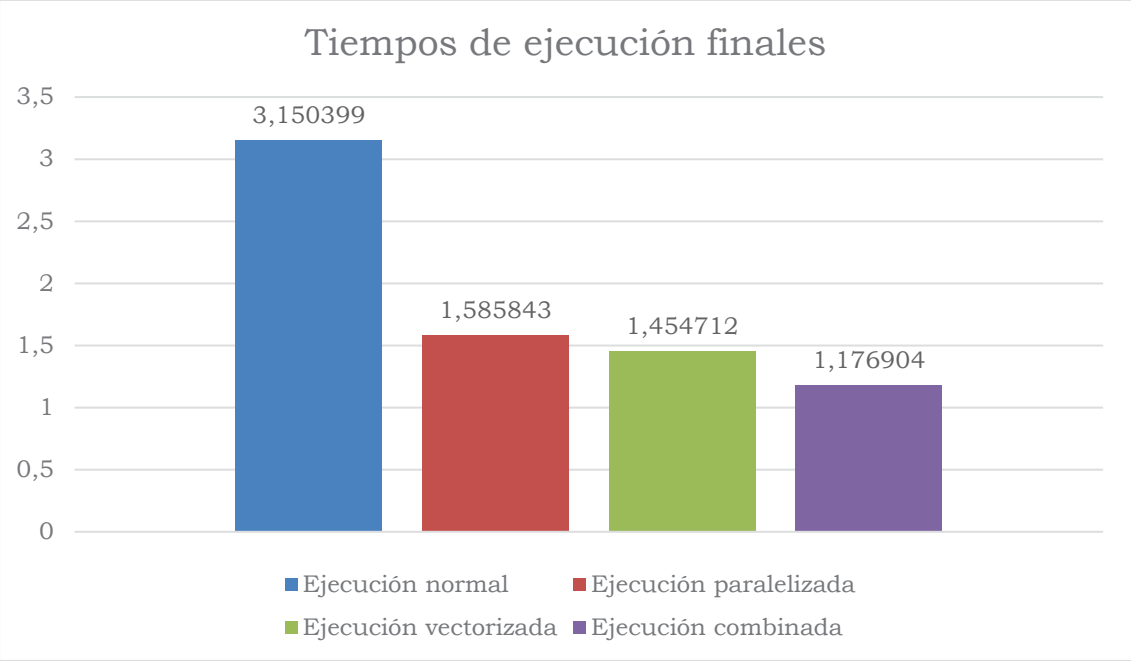
En el gráfico anterior se puede ver el descenso del tiempo de ejecución tras paralelizar las funciones especiales.

3 Resultados y conclusiones

En este apartado se van a tratar los resultados obtenidos tras todo el proyecto y a comentarlos. Además se hará una valoración personal del desarrollo del trabajo y de la asignatura en general, comentando las lecciones aprendidas y la aportación del proyecto a la formación en el campo de la informática.

3.1 Resultados finales

Tras el desarrollo del proyecto, traduciendo, vectorizando y paralelizando la red neuronal, siempre garantizando que los resultados que obtenía esta no se viesen alterados se han obtenido los siguientes resultados finales, los cuales se han obtenido al ejecutar pruebas ligeras (600 imágenes de entrenamiento y 100 de testeo) para poder realizar más pruebas y obtener valores medios en lugar de realizar pocas pruebas grandes y trabajar con menos resultados.



En los resultados obtenidos se puede ver que hay una mejora muy apreciable en los tiempos de ejecución de la red neuronal, llegando a un máximo de 2,6768 veces más rápida. Estos resultados son buenos si se ven de manera aislada, pero en el contexto que ocupa podrían ser bastante mejores.

De manera dividida la paralelización y la vectorización obtienen resultados muy similares, ligeramente mejores los segundos. Esto es sorprendente en primera instancia ya que paralelizar debería ofrecer un rendimiento muy superior teóricamente, pero en la práctica esto no es así debido a múltiples factores.

El principal es que paralelizar es mucho más costoso para el ordenador debido a la sincronización de los múltiples hilos y si no se trata esta sincronización con el debido cuidado puede causar una **gran cantidad de errores**, que es lo que ha llegado a suceder en algunas ocasiones obligando a eliminar regiones paralelas y dejarlas en ejecución secuencial. Además la paralelización alcanza un punto de rendimiento mucho más elevado cuando se tratan operaciones muy numerosas como las ejecutadas en las pruebas explicadas en el apartado correspondiente. En estas pruebas se realizaban millones de operaciones que permiten obtener niveles mucho más altos de ganancia que en operaciones

cortas en las que el coste de crear los hilos de ejecución y sincronizarlos puede llegar a superar a la ganancia obtenida por paralelizarlos.

La vectorización por otro lado ha ofrecido unos resultados muy sorprendentes, tanto por la ganancia obtenida como por la sencillez de aplicarla. Es una tecnología mucho menos conocida, al menos no tanto como la paralelización, y ofrece unos resultados extremadamente positivos con muy poco esfuerzo. Y, aunque también puede llegar a producir una gran cantidad de errores en la ejecución de los programas si no se trata con cuidado, es mucho más difícil que estos se produzcan. El principal cruce en esta tecnología ha sido el de la vectorización manual frente a la vectorización automática, pero al final el mejor resultado lo ofrece el uso combinado de ambas técnicas.

Si se tiene claro lo que se pretende vectorizar se puede lograr muy fácilmente conociendo las técnicas adecuadas y el compilador es capaz de vectorizar operaciones que no se hayan tenido en cuenta a la hora de aplicar la vectorización.

Sin embargo para vectorizar es extremadamente importante la organización de los datos en memoria, problema que también sucede al paralelizar pero en menor medida. Esto puede resultar poco intuitivo para la mayoría de los programadores que están acostumbrados a un estilo común para guardar la información ya que es el más simple y legible, pero si se altera ligeramente la forma de pensar y escribir código se pueden obtener ganancias muy considerables.

También se debe tener en cuenta que a la hora de reservar memoria se debe alinear la información siempre que se pueda ya que si no se hace no se podrá vectorizar, o al menos no se podrán vectorizar todas las operaciones deseadas y se perderá ganancia de rendimiento.

Es extremadamente importante conocer las opciones que ofrecen librerías como OpenMP y como funcionan de fondo, ya que ofrecen muchas soluciones a problemas bastante comunes que en casos de paralelizar o vectorizar manualmente pueden llegar a ser extremadamente costosos, como acumulación de resultados, división de trabajo en bloques o sincronización.

Finalmente se concluye que, aunque en este caso los resultados podrían llegar a ser mejores si se le dedicase más tiempo, se han obtenido buenas mejoras en los tiempos de ejecución y no ha hecho falta aplicar la cantidad de esfuerzo enorme que típicamente suponen los problemas de optimización mediante estas tecnologías.

3.2 Opiniones y valoración

En lo personal el desarrollo del proyecto ha sido un poco más complicado de lo esperado debido a la situación vivida por la pandemia del Covid-19 y el confinamiento. Esto sin duda ha generado dificultades para organizarse y avanzar, pero se ha seguido trabajando constantemente y se ha llegado a concluir.

En lo que respecta a la parte más técnica la mayor dificultad y que menos esperaba que sucediese ha sido la traducción de la red neuronal de Python a C. Aunque aparentemente puede resultar una tarea sencilla, especialmente teniendo en cuenta lo corta que era (a penas 140 lineas) ha sido bastante difícil traducirla a C. La gestión de memoria es el principal problema y ha sido el que, probablemente, más tiempo ha ocupado del desarrollo. Se han cometido errores y se han aprendido de ellos, considero que refuerzos como este en lenguajes de programación de más bajo nivel son esenciales para entender el funcionamiento

real de lo que se hace. Sin embargo también se entiende perfectamente por qué estos lenguajes cada día son menos comunes, especialmente en la industria general. Su coste de desarrollo es mucho más elevado al requerir más tiempo por línea y no disponer de tantas librerías auxiliares que faciliten el trabajo. Además aportan un factor muy importante al resultado final, la legibilidad del código. Resulta muchísimo más sencillo leer un código escrito en Python en C al ser mucho más simple y directo, pero si se hace bien se pueden obtener resultados muchísimo mejores en cuanto a tiempos de ejecución y eficiencia de memoria si se emplean adecuadamente los lenguajes de más bajo nivel.

En cuanto al aprendizaje personal que ha aportado el proyecto ha sido inmenso. Reforzar el manejo de Python y C es extremadamente útil ya que el primero comencé a usarlo hace poco y el segundo llevaba mucho sin tocarlo. Pero sobre todo valoro el aprendizaje sobre las tecnologías de paralelización y vectorización. Saber que existen sistemas para facilitar la aplicación de estas técnicas de manera que se le pueda sacar el máximo rendimiento a un ordenador sin tener que realizar operaciones extremadamente complejas es una lección muy valiosa. Y directamente desconocía la existencia de la vectorización y las operaciones vectoriales, de las cuales me llevo la mejor opinión puesto que han ofrecido un rendimiento excelente con muy poca dificultad.

Pero la mayor lección aprendida es para el diseño futuro de software. Aunque estas librerías faciliten mucho la aplicación de estas tecnologías es extremadamente importante orientar el diseño de los programas a su uso desde el principio. Una de las mayores dificultades ha sido corregir las dependencias de datos existentes que impedían la paralelización y vectorización y también realizar las reservas de memoria alineadas sin que llenasen la memoria disponible ni ralentizasen más de lo que podían llegar a acelerar.

4 Bibliografía

[1] OpenMP, The OpenMP API specification for parallel programming, 1997. [Online] Available: https://www.openmp.org/

[2] M. A. Nielsen, Neural Networks and Deep Learning, Determination Press, 2015. [Online] Available: http://neuralnetworksanddeeplearning.com/

[3] Y. LeCun, C. Cortes, C. J. C. Burges, The Mnist Database, 1998. [Online] Available: http://yann.lecun.com/exdb/mnist/

[4] F. Rosenblatt, Principles of neurodynamics: perceptrón and the theory of brain mechanisms, Michigan: Spartan Books, 1962

[5] J. Deslippe, H. He, H. Wasserman, W. S. Yang, OpenMP and Vectorization Training Introduction, 2016. [Online] Available: https://www.nersc.gov/assets/Training-Materials/NERSC-VectorTrainingOct2014.pdf

[6] OpenMP, The OpenMP API Specification: Version 5.0, 2018. [Online] Available: https://www.openmp.org/spec-html/5.0/openmpsu42.html

[7] G. Román Díez, Condiciones de Carrera, 2019. [Online] Available: https://babel.upm.es/teaching/concurrencia/material/slides/groman/CC_CondCarrera.pdf

[8] U.S. Department of Energy, Introduction to Parallel Computing, 2020 [Online] Available: https://computing.llnl.gov/tutorials/parallel_comp/

[9] X. Martorell, Parallel Programming with OpenMP, 2012 [Online] Available: https://www.bsc.es/sites/default/files/public/computer_science/extreme_computing/omp_tutorial.pdf

[10] Universidad Carlos III, Curso de OpneMP, 2020 [Online] Available: http://ocw.uc3m.es/ingenieria-informatica/arquitectura-de-computadores-ii/otros-recursos-1/or-f-008.-curso-de-openmp

[11] B. Kernighan, D. Rictchie, The C programming Language, 1988

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Fri Jul 10 10:56:11 CEST 2020
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)