Paralelización de redes neuronales con OpenMP

Ignacio Rojo Pérez y Marcel Sarraseca Julián

Marzo 2021

1 Introducción

El objetivo de esta experimentación es la paralelización de una red neuronal y optimizar su rendimiento mediante la explotación del paralelismo MIMD (Multiple Instructions Multiple Data) a través de OpenMP. El código se ejecutará en una máquina de memoria compartida y mediante OpenMP se buscará la máxima explotación de los recursos brindados para mejorar su rendimiento.
Se realizará un análisis para saber qué funciones son las que más porcentaje del tiempo de ejecución consumen, y posteriormente se realizarán las modificaciones pertinentes en el código para tratar de reducir su tiempo de ejecución a través de la librería de OpenMP, la cual permite la paralelización del programa usando múltiples threads en múltiples CPUs.

El algoritmo a trabajar se divide principalmente en dos funciones: TrainN() y runN().

La función TrainN() se encarga de entrenar la red neuronal con un conjunto de patrones, para que posteriormente la red pueda clasificar un conjunto de datos mediante la función runN().

2 Análisis

Previamente a emplear una estrategia de optimización y tratar de paralelizar el algoritmo dado, se deben analizar que funciones que son las que más tiempo de ejecución están consumiendo. Estas funciones serán el principal objetivo a tratar, ya que por un lado serán las que mayor rendimiento proporcionaran y por el otro, se hallaran los factores limitantes o cuellos de botella de la aplicación.

Utilizando la herramienta de perf, ejecutando "perf record ./ejecutable" y luego "perf report" se observa que la función TrainN(), consume el 95,97% del tiempo de ejecución. El 3,63% se atribuye a la asignación de librerías por parte del sistema operativo y el 0,04 % restante a las demás funciones del algoritmo.

En este caso, la función runN() no tiene ninguna relevancia por lo que la prioridad a tratar será la función Train(). Más adelante, si runN() tiene relevancia, se procederá a su debido tratamiento.

3 Paralelización del algoritmo

Antes que nada, es necesario comentar la forma en que se va a tratar el problema. Este problema se puede tratar de dos formas: la primera, que cada thread se encargue de un solo bucle (en general) a la vez y la segunda, donde cada bucle sea tratado por todos los threads a la vez. Se ha elegido la segunda, debido a que de forma sencilla se puede lograr un mejor balanceo de carga entre threads.

En este caso se han paralelizado los bucles de mayor peso computacional, muchos de ellos con la directiva #pragma omp for. Esta directiva permite repartir entre los threads "x" iteraciones de forma equitativa. Muchas de las estructuras del programa tienen un patrón de cómputo tipo MAP, esto quiere decir que las iteraciones son independientes unas de las otras y no siguen un orden concreto, el resultado será el mismo, con lo cual es muy conveniente emplear esta directiva que, además de asignar "x" iteraciones por thread, balancea la carga de trabajo de forma automática.

En algunos casos se ha empleado la directiva #pragma omp single para permitir que un único thread realice la ejecución de determinado fragmento del código, esto es así debido a que la interacción de varios threads escribiendo en una variable compartida darían lugar a data races y el resultado seria incorrecto.

Para optimizar la función TrainN, se ha decidido crear una región paralela con variables compartidas y variables firstprivate para que cada hilo tenga su propia instancia con el valor anterior a la declaración de la región paralela, situada después de la función LoadPatternSet.

```
if ((tSet = loadPatternSet(NUMPAT, "optdigits.tra", 1)) == NULL) {
    printf("Loading Patterns: Error!!\n");
    exit(-1);
}
#pragma omp parallel shared (trobat, Hidden, Output, DeltaO, BError, Error, DeltaH,DeltaWeightIH,ranpat,DeltaWeightHO) firstprivate(SumH, SumO, SumDOW)
{
    for (int i = 0; i < NUMHID; i++)
        for (int j = 0; j < NUMIN; j++) {
            WeightIH[i][j] = 2.0 * (frando() + 0.01) * smallwt;
            DeltaWeightIH[i][j] = 0.0;
        }

    for (int i = 0; i < NUMOUT; i++)
        for (int j = 0; j < NUMHID; j++) {
            WeightHO[i][j] = 2.0 * (frando() + 0.01) * smallwt;
            DeltaWeightHO[i][j] = 0.0;
        }
```

Figure 1: Directiva parallel shared y firstprivate

Dentro del bucle principal que se encarga de iterar las weight updates, se ha paralelizado el siguiente bucle interno con un patrón de cómputo MAP usando la directiva #pragma omp for para que los hilos disponibles se repartan el trabajo equitativamente

Para el bucle a continuación se ha optado por la directiva #pragma omp single, ya que se están realizando accesos aleatorios a memoria y si se paralelizase podría darse el caso que las posiciones seleccionadas estén asignadas a threads distintos.

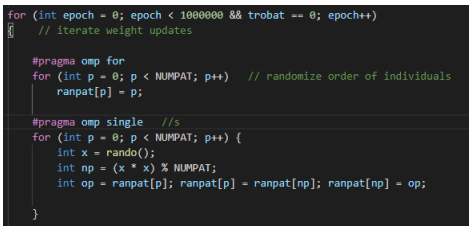

Figure 2: Directivas omp for y single

Dentro del bucle que itera sobre cada batch, se ha decidido incluir otro #pragma omp for sobre el segundo bucle más interno donde cada thread tendrá su variable privada SumH y conseguirán repartirse los accesos al vector Hidden.

Posteriormente se realiza un #pragma omp for reduction(+:BError) en el siguiente bucle. Con esta directiva lo que se consigue es realizar un sumatorio de la variable BError. Cada thread crea su variable privada BError y al finalizar la ejecución del bucle se realiza un sumatorio total de BError de cada thread individual.

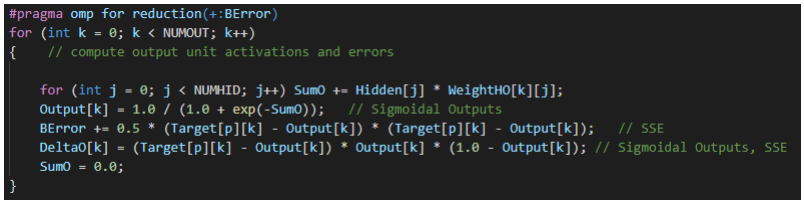

Figure 3: Directiva reduction

Los dos penúltimos bucles se han paralelizado usando la directiva #pragma omp for collapse(2) que consigue unir los bucles y repartir las iteraciones del bucle interno y externo de manera equitativa.

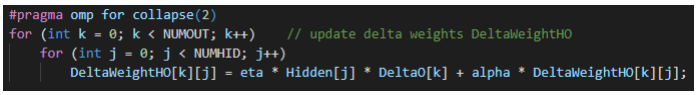

Figure 4: Directiva collapse

Por último, se ha empleado la directiva #pragma omp single al imprimir por pantalla el "epoch" y el "Error", de tal forma que sea un único thread que realize dicha tarea. En este caso la instrucción "break" inicialmente en el código para romper el bucle daba problemas al insertar la directiva mencionada. Entonces se ha optado por emplear una variable auxiliar llamada "trobat" (inicializada

a zero) y añadir una condición adicional en el bucle externo (epoch), de esta forma cuando Error ¡ 0.0004, se modificará el valor de "trobat" a uno y por ende se saldrá del bucle, evitando usar la instrucción "break" y permitiendo emplear la directiva de openMP tal y como se ha planteado.

```
for (int epoch = 0; epoch < 1000000 && trobat == 0; epoch++)
{    // iterate weight updates
```

Figure 5: Doble condición con la variable "trobat"

```
#pragma omp single
{
    Error = Error / ((NUMPAT / BSIZE) * BSIZE); //mean error for the last epoch
    if (!(epoch % 100)) printf("\nEpoch %-5d :   Error = %f \n", epoch, Error);

    if (Error < 0.0004)
    {
        trobat = 1;
        printf("\nEpoch %-5d :   Error = %f \n", epoch, Error);   // stop learning when 'near enough'
    }
}
```

Figure 6: Implementación de la variable "trobat" junto con la directiva de openMP

4 Resultados

Partiendo de la versión original compilada con -Ofast ejecutada de manera secuencial se obtiene un tiempo de ejecución de 325.63s con un sobrecoste mínimo, probablemente atribuido a la invocación de las funciones o la propia representación de la información en el fichero de salida.

Duplicando la cantidad de recursos disponibles, la ejecución del algoritmo se consigue una mejora del 84% frente al original. En cuanto al overhead esta vez se puede decir que ha aumentado significativamente debido a los procesos de comunicación entre threads.

Como se puede ver, en términos de escalabilidad, para un problema de tamaño fijo como es el que se presenta, a medida que se aumenta el número de procesadores, el tiempo de ejecución disminuye, pero el tiempo de comunicaciones aumenta. A partir de una cantidad determinada de procesadores los tiempos de cálculo serán inferiores a los de comunicación dando como resultado una ejecución poco eficiente.

5 Conclusiones

Cabe destacar que se ha cumplido con el objetivo de la práctica. Se ha logrado aplicar paralelismo MIMD a la red neuronal mediante el uso de la librería de OpenMP, logrando una versión funcional y más eficiente que la versión original

Recursos (CPUs) i Threads	Temps (s)	Cost (s)	Eficiència	Speedup	Overhead
1	325.62	325.37	-	-	0.25
2	176.82	353.33	0.92	1.84	27.71
4	94.6	377.65	0.86	3.44	52.03
6	70.45	421.84	0.77	4.62	96.22
8	60.5	482.85	0.67	5.38	157.23
12	52.65	630	0.52	6.18	304.38

Figure 7: Taula de resultats

single thread compilada con "-Ofast". La mejora obtenida entre la versión secuencial compilada con -Ofast y la versión paralela, con 12 threads/12 Cores es de 6.18x. Aunque la mejora es significativa, se tienen que tener en cuenta otros parámetros para determinar la calidad de la solución hallada.

En la tabla de resultados se observa cómo va descendiendo la eficiencia de la solución propuesta y como va aumentando el overhead a medida que se van añadiendo más recursos de cómputo. Esto sucede debido a que al aumentar de recursos, estos requieren de mayor comunicación entre sí y esto se traduce en un mayor costo en comunicaciones y por ende, empeora la eficiencia.

En cuanto a escalabilidad, la solución propuesta presenta limitaciones al añadir recursos, si se añadiesen muchos recursos, se llegaría a un punto en el que el tiempo de comunicaciones sería mayor al tiempo de cómputo, lo que daría lugar a una ejecución ineficiente.

Una posible solución ante las limitaciones en cuanto a escalabilidad que presenta la memoria compartida y OpenMP, seria emplear un lenguaje de paso de mensajes como MPI. Este lenguaje brinda la posibilidad de emplear nodos de forma distribuida y realiza las comunicaciones mediante el paso de mensajes, esta forma de comunicación es más eficiente que las comunicaciones en memoria compartida y no sobrecarga tanto la red de interconexión.

6 Anexos

6.1 Compilación y ejecución del programa

Antes que nada, para realizar correctamente la compilación del programa se requieren de los siguientes archivos: nn-main.c common.h, common.c, optdigits.cv y optdigits.tra.

Primero se actualiza a la versión más reciente del compilador gcc, en nuestro caso se empleará la versión gcc/10.2.0 con la sentencia "module add gcc/10.2.0".

En cuanto a la compilación del código, quedaría de la siguiente forma: gcc -fopenmp -std=c99 -Ofast -lm common.c nn-main.c -o nnmain.exe

La opción "-fopenmp" permite habilitar las directivas de OpenMP en el código. Con "std=c99" se le indica al compilador que se va a emplear la versión

estándar C99. La opción "-Ofast" es el máximo grado de optimización que realiza el compilador al crear el ejecutable del programa. Y la opción "-lm" permite incluir la librería math.

Para enviar el ejecutable a Wilma se ha empleado la siguiente sentencia: sbatch -o resultado.txt ./script.sh nnmain.exe

En resultado.txt se almacenarán los resultados de la ejecución: Ejecución del programa juntamente con los resultados de la herramienta "perf", a través del script.sh se envía el ejecutable "nnmain.exe" al clúster Wilma a través de SLURM.

6.2 Script SLURM

```
#!/bin/bash
#SBATCH -N 1 #N num nodes
#SBATCH -n 1 #n num tasks
#SBATCH –exclusive

   export OMP NUM THREADS=12
perf stat $1 $2 $3 $4
```

References

[1] OpenMP
Technical Information [Online].Available:
https://www.openmp.org/wp-content/uploads/OpenMPRefCard-5.1-web.pdf

[2] Bullinaria, J., Implementing a neural network in C. Available: https://www.cs.bham.ac.uk/ jxb/NN/nn.html.