Paral·lelització de xarxes neuronals

Blanca Llauradó Crespo

Resum– Aquest treball té per objectiu la modificació i l'optimització d'una xarxa neuronal, escrita en C, de manera que, a partir de tres aproximacions d'HPC (High-Performance Computing), concretament *OpenMP*, *MPI* i *CUDA*, s'incrementi el seu rendiment pel que fa al temps d'execució. En aquest article es fa un estudi de la paral·lelització de la xarxa utilitzant les tres aproximacions esmentades anteriorment, modificant les estructures d'emmagatzematge de dades i l'obtenció dels patrons d'entrenament. D'aquesta manera, per una banda, es decrementa el temps destinat a l'entrenament de la xarxa. Per l'altra, s'incrementa la fiabilitat dels resultats, ja que s'utilitzarà un entrenament de patrons aleatori.

Paraules clau– Xarxes neuronals, *Feedforward*, *Backpropagation*, HPC, *OpenMP*, *MPI*, *CUDA*, aplicacions paral·leles, optimitzacions.

1 INTRODUCCIÓ

UNA xarxa neuronal és un model d'aprenentatge i processament automàtics que emula el comportament de la ment humana a l'hora de la resolució de problemes comuns. HPC, en canvi, són aplicacions que realitzen operacions intensives aprofitant les arquitectures hardware, a través del processament paral·lel.

Aquest treball està destinat a l'optimització d'una xarxa neuronal utilitzant tres aproximacions HPC diferents -segons els recursos de què es disposi-, així com a la descripció del seu funcionament i actuació.

Entre les diferents maneres d'optimitzar un codi, hi podem trobar la utilització del paral·lelisme que proporciona la repartició de la càrrega de feina entre els diversos Cores de què disposi el processador -sempre suposant que el codi sigui paral·lelitzable-, la divisió de la càrrega entre diversos nodes, o l'aprofitament del paral·lelisme que ofereixen els CUDA Cores dels quals disposi la GPU. Tot dependrà de les prestacions que es tinguin a l'abast.

Així doncs, els objectius principals d'aquest treball són: 1) Analitzar el codi de la xarxa escollida en C, així com el seu rendiment; 2) La seva adaptació per tal que sigui paral·lelitzable amb *OpenMP*, *MPI* i *CUDA*, tres aproximacions que permeten una acceleració del codi segons els mitjans dels quals es disposi i que requeriran d'una refactorització del codi; 3) El desenvolupament, l'optimització i l'avaluació del rendiment de la xarxa neuronal proposada utilitzant les tres aproximacions utilitzades a HPC i, 4) La comparació de l'efecte, respecte al rendiment, de cadascuna de les aproximacions.

D'aquesta manera, en la secció 2 es presenta l'estat de l'art. En la secció 3 es descriuen les xarxes neuronals, dividint aquesta explicació entre una descripció general, el funcionament de la xarxa -concretament a la secció 3.1-, i els tipus que n'existeixen -exposats a la secció 3.2-. A continuació, a la secció 4 s'introdueix el concepte d'HPC i a la secció 5 es mostra la relació entre les xarxes neuronals i l'HPC amb aquest treball. La secció 5 ja s'endinsa en el desenvolupament del treball en sí, presentant les modificacions que s'han realitzat al codi per a utilitzar dades consecutives en memòria i la necessitat de la barreja de patrons. Finalment, les seccions 6, 7 i 8 se centren en la paral·lelització de la xarxa amb *OpenMP*, *MPI* i *CUDA* respectivament, mostrant els canvis efectuats en el codi i els resultats obtinguts.

Pel que fa a la metodologia seguida, s'ha utilitzat una metodologia *agile*, la qual permet anar atenyent els objectius per iteracions i corregint els errors i la planificació a mesura que aquests sorgeixen, sense perjudicar el treball global ja planificat.

2 ESTAT DE L'ART

Les xarxes neuronals van aparèixer a la dècada del 1950. Tanmateix, no es van popularitzar degut a la baixa potència que tenien els equips en aquell moment, així com a la inexistència de bons algoritmes d'entrenament. Actualment, gràcies als algoritmes de *Backpropagation* –algorisme d'aprenentatge mitjançant descens de gradients–, a l'ús de les GPUs –que permeten computar grans quantitats de càlculs– i a les millores de les bases d'entrenament, les xarxes neuronals han tornat a adquirir un paper important en nombrosos camps, principalment centrats en tasques com la predicció i la classificació. Això ha permès, a més a més, l'aparició del *Deep Learning* –que utilitza xarxes neuronals profundes–[1].

Alguna de les àrees que s'han vist beneficiades per aquesta tecnologia són la Indústria 4.0, l'economia –on es pot predir, per exemple, quant variaran els preus dels productes amb els anys–, la medicina –per a diagnosticar problemes de salut–, el reconeixement i la classificació, el processament de dades i la modelització –amb disseny i búsqueda d'errors en sistemes de software complexos–, l'enginyeria

• E-mail de contacte: blancamercedes.llaurado@autonoma.cat
• Treball tutoritzat per: Anna Bàrbara Sikora
• Curs 2021/22

de control –amb la monitorització de sistemes informàtics i la manipulació de robots– i la intel·ligència artificial –utilitzant el *Deep Learning* i el *Machine Learning*–[2].

3 XARXES NEURONALS

Es coneix com a *Machine Learning* a una subdivisió de la intel·ligència artificial que utilitza els algoritmes que aprenen constantment i que gira al voltant dels sistemes informàtics que analitzen les dades i n'aprenen [3]. Així doncs, el *Machine Learning* és l'enfocament alternatiu per al disseny d'una solució algorítmica que, en l'enginyeria convencional, suposaria un gran problema [4].

Per la seva banda, es coneix com a *Deep Learning* el model computacional, compost de múltiples processos distribuïts en capes, que permet aprendre representacions de dades amb múltiples nivells d'abstracció [5].

Aquests camps d'estudi i aprenentatge es valen d'un paradigma d'aprenentatge i processament automàtics anomenat xarxa neuronal, la qual reflecteix el comportament de la ment humana, permetent als programes reconèixer patrons i solucionar problemes comuns en l'àmbit de la intel·ligència artificial"[6].

Aquestes xarxes neuronals estan, doncs, com el sistema nerviós humà, compostes per neurones –unitats de processament–, unides en el que es coneix com a capes, que funcionen de manera simultània i interconnectada.

Pel que fa a les capes, n'existeixen tres tipus, segons el lloc on es troben i la funció que tenen. Tenim, inicialment, la capa d'entrada, la qual conté les neurones que s'utilitzaran per representar els camps d'entrada (*input layer neurons*). Seguidament, es troben les capes ocultes (*hidden layers neurons*) que contenen les unitats no observables a través de les quals es van propagant els valors i les connexions des de cada neurona a la següent capa fins a trobar el resultat final. I, finalment, hi ha la capa de sortida (*output layer neurons*) que representa el camp o camps de destí.

La Figura 1 esquematitza l'estructura d'una xarxa neuronal de tres capes: entrada (*input*), oculta (*hidden*) i sortida (*output*).

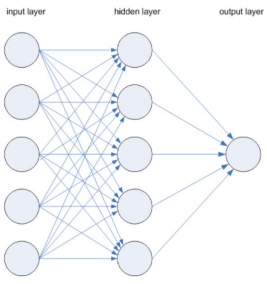

Fig. 1: Estructura d'una xarxa neuronal [6]

3.1 Funcionament d'una xarxa neuronal

Per conéixer el funcionament intern d'una xarxa neuronal, el primer que cal saber és que cada neurona de cada capa conté els seus pesos (w) o ponderacions, el seu valor bias -que permet aconseguir una convergència més ràpida de la xarxa desplaçant la funció d'activació [7] i [8]- i la sortida, que serà transmesa a la següent capa en el cas que aquesta neurona s'activi –és a dir, que el resultat estigui per d'un valor llindar–. Els pesos contribueixen de manera més significativa a la sortida en el cas de ser majors del valor llindar, la qual cosa ajuda a determinar la importància de la variable. Finalment, totes les entrades de cada capa es multipliquen pels seus pesos i se sumen.

Per tant, tot i que l'entrenament és indispensable en una xarxa neuronal, el que de veritat permet el seu funcionament és la manera com, internament, s'actualitzen els pesos i es crea una relació entre la entrada i la sortida.

La fórmula, per a calcular el valor de cadascun dels nodes seria, de forma simplificada, la següent:

$$\sum_{i=1}^{m} w_i x_i + bias = w_1 x_1 + w_2 x_2 + w_3 x_3 + ... + w_m x_m + bias$$

on la variable *m* fa referència al nombre de neurones de la capa, la variable *w* al pes de la neurona, la variable *x* al valor d'activació o d'entrada i el bias, com ja s'ha esmentat, al valor de desplaçament.

La sortida passa a una funció d'activació, que mira si el valor és superior al llindar per a deixar-lo passar a la següent neurona de la següent capa, seguint l'esquema matemàtic següent [9]:

$$output = f(x) = \begin{cases} 1 \; if \sum w_i x_i + bias \geq 0 \\ 0 \; if \sum w_i x_i + bias < 0 \end{cases}$$

Tot i així, cal destacar que hi ha altres mètodes que no impliquen l'obligatorietat d'haver d'obtenir un 0 o un 1 al valor de sortida de la neurona segons el resultat del càlcul anterior, sinó que aquest valor pot ser calculat, de manera més complexa, seguint unes altres directrius, com per exemple, aplicant la funció ReLU (Rectified Linear Unit) [10].

3.2 Tipus de xarxes neuronals

Segons l'objectiu per al qual es vulgui utilitzar la xarxa neuronal, tenim tres tipus de xarxes: els perceptrons multicapa o xarxes neuronals feedforward (MLP) –les quals s'han esmentat fins al moment–, les xarxes neuronals convolucionals (CNN) i les xarxes neuronals recurrents (RNN).

Les xarxes feedforward es componen d'una capa d'entrada, N capes ocultes i una capa de sortida, i són la base de la visió per computador, el processament del llenguatge natural i d'altres xarxes neuronals. Cal tenir en compte, però, que donat que la majoria de problemes a resoldre no són lineals, normalment estan compostes per neurones sigmoides, no per perceptrons [9].

La funció sigmoide és la funció d'activació més antiga i popular i es defineix com [11]:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Els perceptrons, en canvi, s'utilitzen per als patrons linealment separables, és a dir, els que es troben a ambdós costats d'un hiperplà [12].

Per la seva banda, les xarxes neuronals convolucionals s'acostumen a utilitzar per al reconeixement d'imatges, patrons i/o la visió per computador, i utilitzen la multiplicació de matrius per identificar patrons dins d'una imatge, aprofitant principis d'àlgebra lineal.

Finalment, les xarxes neuronals recurrents s'utilitzen en àmbits com ara el mercat de valors o els pronòstics de vendes, ja que fan servir bucles de retroalimentació per fer prediccions sobre valors futurs.

En resum, tots els tipus de xarxes, independentment de les seves diferències pertinents, són complexos i poden trigar un gran temps en entrenar-se i en reconèixer, posteriorment, els patrons.

4 HPC (HIGH-PERFORMANCE COMPUTING)

Les aplicacions que utilitzen HPC s'aprofiten de les arquitectures hardware i software per tal de realitzar operacions intensives, fent servir el processament paral·lel d'aquestes, i així incrementar el seu rendiment i escalabilitat. D'aquesta manera, les aplicacions poden ampliar la computació en múltiples sistemes fent que actuïn com un de sol. És per aquest motiu que s'utilitzen a l'hora d'optimitzar el rendiment de programes complexos [13].

La paral·lelització consisteix a convertir un codi seqüencial per tal d'utilitzar múltiples processadors simultanis en una màquina amb multiprocessadors de memòria compartida i distribuida [14]. D'aquesta manera, es volen dividir tasques per ser realitzades de forma paral·lela. Aquest paral·lelisme es pot aconseguir a nivell de procediment –és a dir, realitzant diverses crides a procediments de forma simultània–, a nivell de bucles –on s'executen diverses iteracions del bucle en paral·lel– o a nivell de bloc –on s'executen diverses operacions d'un bloc a la vegada–[15].

Per norma general, la major part del consum d'un programa es troba en els bucles. És per això, que, en aquest treball, es prioritza la paral·lelització dels bucles de la xarxa neuronal utilitzant tres aproximacions diferents: **OpenMP**, **MPI** i **CUDA**. Cal saber, però, que un codi no és mai completament paral·lelitzable. Això és degut al fet que sempre hi ha una part del càlcul del codi que és seqüencial, cosa que provoca que, en el cas que F sigui la fracció d'un càlcul seqüencial, només (1-F) serà la fracció que pot ser paral·lelitzada. Aleshores, la màxima acceleració (speedup) que pot aconseguir el programa serà de:

$$\frac{1}{F + (1 - F)/N}$$

sent N el nombre de processadors. Aquesta relació es coneix com a llei d'Amdahl [16].

OpenMP és una interfície de programació d'aplicacions que suporta programació multiprocés amb memòria compartida fent una implementació multithreading, és a dir, utilitzant el principi master-team, on el thread "master" es bifurca en threads "team" per tal de dividir la tasca i poder-la realitzar de forma concurrent. Aquesta interfície està composta d'un conjunt de directives de compilador que afecten al comportament de l'algoritme en temps d'execució, i és molt utilitzada en llenguatges com Fortran C o C++ [17].

MPI, per la seva banda, és una especificació d'interfície de biblioteca de programació, basada en la comunicació a través de pas de missatges. Actualment és utilitzada en la programació concurrent fent servir memòria distribuida, i les operacions són expressades com a funcions, subrutines o mètodes enllaçats fent servir API de la llibreria desenvolupada i optimitzada [18]. Utilitzant aquesta interfície és molt important, però, controlar la quantitat de missatges que s'aniran enviant i si la comunicació es farà de manera col·lectiva –involucrant comunicacions entre tots els processos creats– o punt a punt –on la comunicació es realitza, exclusivament, entre els dos processos vinculats–.

Finalment, **CUDA** és una plataforma de computació paral·lela de memòria compartida i model de programació d'aplicacions que permet accelerar l'execució dels codis fent servir Unitats de Processament Gràfic (GPU) creades per Nvidia. D'aquesta manera, el codi es divideix en kernels, els quals utilitzen la jerarquia de thread, bloc i grid per dur a terme la paral·lelització. Cal saber, però, que la memòria dels threads és privada i d'ús exclusiu. En canvi, la memòria dels blocs és visible per a tots els threads del bloc, i la memòria global és accessible per a tots els threads [19].

Amb totes aquestes interfícies, l'objectiu final és millorar el rendiment del programa -en aquest cas la xarxa neuronal- pel que fa al temps d'execució de l'algoritme.

5 XARXA ORIGINAL I MODIFICACIONS

Per a aquest treball és necessari partir d'una xarxa neuronal prou complexa com per poder ser optimitzada utilitzant les interfícies comentades en el punt anterior. Inicialment, es van buscar i analitzar quatre possibles opcions de xarxes que podien servir com a punt de partida: una xarxa neuronal implementada en C però massa simple per al propòsit d'aquest treball [20], una de més complexa però que no s'acabava d'ajustar al dessitjat per aquest treball [21], una que complia els requeriments de complexitat però que estava programada en C++ -cosa que implicava haver d'utilitzar primitives de OpenMP, MPI i CUDA específiques per a C++-[22] i, finalment, una quarta opció.

Aquest treball parteix d'aquesta última proposta, una xarxa neuronal del tipus *feedforward* o perceptrons multicapa -la qual utilitza el mètode d'entrenament *Backpropagation*- programada en C [23], composta per una capa neuronal d'entrada (input layer neuron), capes neuronals amagades (hidden layers neuron) –en les quals es pot escollir la quantitat dessitjada– i una capa neuronal de sortida (output layer neuron). A més a més, per a totes les capes també es pot indicar el nombre de neurones.

A partir d'aquí, s'analitza el comportament d'aquesta xarxa, és a dir, el temps que triga en executar-se de manera seqüencial i sense utilitzar cap optimització. El comandament que s'ha utilitzat per a executar inicialment l'algoritme és el següent:

gcc main.c layer.c neuron.c common.c -o main –lm

fent servir un training dataset per al reconeixement de patrons numèrics utilitzat amb anterioritat a l'assignatura de Computació d'Altes Prestacions.

Una xarxa neuronal –tal i com s'ha esmentat en la secció 3– necessita d'una base d'entrenament per tenir èxit en la resolució del problema. Tanmateix, com s'ha mostrat en la secció 3.1, el bon funcionament de la xarxa no només es basa en el training dataset que s'utilitzi per tal que l'algoritme aprengui, sinó en la manera com internament actualitza el seu model utilitzant pesos (weights) per crear un mapeig i, per tant, una relació entre la informació d'entrada i la de sortida.

Aquesta utilització dels pesos consisteix a trobar el valor que millor soluciona el problema fent ús d'un procés

iteratiu, on els pesos van variant lleugerament fins a trobar el valor final, és a dir, el que proporciona el mínim error a l'hora d'utilitzar el conjunt d'entrenament (training dataset).

Així doncs, entrenar una xarxa neuronal és un procés complicat, i optimitzar-la a nivell software, un repte, ja que s'han de tenir en compte els casos on la superfície d'error no és convexa i conté mínims locals, o taques planes, ja que llavors l'algoritme no pot tenir clar quin ha de ser el seu següent moviment.

És per aquest motiu, que, en aquest treball, l'optimització del codi s'aborda des del punt de vista de la programació paral•lela, és a dir, des d'un punt de vista hardware, utilitzant els principis que proporciona HPC, per tal de millorar el temps d'execució.

5.1 Implementació de la xarxa

Abans d'entrar en l'explicació de les modificacions efectuades en el codi, cal, però, tenir una visió global de la xarxa que es treballarà.

Aquesta, inicialment, permet indicar el nombre de capes i de neurones per capa que es desitja, així com la taxa d'aprenentatge i el nombre de patrons d'entrenament que s'utilitzaran. Les funcions efectuades per capa i neurona són: 1) *feed_input()* -encarregada d'omplir l'atribut *actv* de cada neurona amb l' *input* corresponent-, 2) *forward_prop()* -encarregada de propagar el valor de la neurona a la següent capa en cas d'acomplir els requisits necessaris-, 3) *compute_cost(p)* -obtenir el cost total de la xarxa-, 4) *back_prop(p)* -actualitzar de nou els valors des del final al principi utilitzant la tècnica de *Backpropagation*- i, finalment, 5) *update_weights()* -actualitzar els pesos per a la propera iteració-.

L'estructura inicial del programa consta d'un *array d'structs -lay[i]-* on cada element de l'*array* és una de les capes de la xarxa neuronal. Cadascun d'aquest *structs* consta d'un *integer* -que indica el nombre de neurones de la capa- i d'un apuntador a un nou *struct*, de manera que es crea un nou *array* de *structs* que representa totes les neurones de la capa. Finalment, per a cada neurona hi ha 6 atributs *floats* i dos apuntados a *float*. Aquest dos últims fan referència als *arrays* dels pesos de cada neurona *-out_weights* (pesos actuals de cada neurona) i *dw* (modificació d'aquests pesos)-. L'esquema es pot observar en la Figura 2.

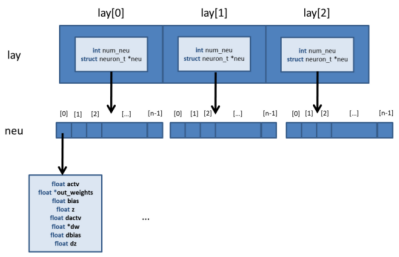

Fig. 2: Esquema inicial de la xarxa neuronal

L'explicació anterior demostra que els accessos inicials als atributs de cada neurona no estan consecutius en memòria.

5.2 Modificació del codi per utilitzar dades consecutives en memòria

Per tal de poder paral•lelitzar un codi independentment del mètode utilitzat, s'aconseguirà, teòricament, un rendiment major si les dades estan consecutives en memòria.

En el cas d'utilitzar OpenMP, els threads es distribueixen entre ells les diferents iteracions d'un bucle. Així doncs, si les dades estan consecutives en memòria, quan un thread accedeix a una dada, s'emportarà a la memòria cache del processador el bloc que conté, no només aquesta dada, sinó també les següents. Si la dada posterior no es trobés consecutiva en memòria, per a tractar-la a la següent iteració, seria necessari accedir de nou a la memòria principal, la qual cosa comportaria un increment significatiu en el temps d'execució del programa. En el cas d'estar ja al mateix bloc que la dada predecessora, a la següent iteració, el thread només haurà d'accedir a la informació requerida que ja es trobarà en la memòria cache. Això comportarà que el temps d'accés sigui menor.

En el cas de MPI, el raonament principal rau en el fet que el cost d'enviar múltiples dades en un únic missatge és menor que haver-les d'enviar en diversos, ja que, en el segon cas, la latència de comunicació entre els processos augmenta [24].D'aquesta manera, sembla una bona praxis intentar que tota la informació necessària per a cada procés ja es trobi emmagatzemada de forma consecutiva. Tot i així, per tal de fer el programa i la distribució de les dades més flexible, MPI també compta amb primitives per als casos en els quals les dades estiguin disperses en la memòria. Això no obstant, les col·lectives sí que funcionen amb memòria consecutiva.

Per últim, CUDA proporciona un accés eficient a memòria, mitjançant el que es coneix com a warps -bloc de 32 fils, que executen la mateixa instrucció-.

S'ha d'aprofitar la lectura i escriptura de cadascun dels 32 elements implicats, de manera que aquests es trobin consecutius en memòria, independentment que es tracti de memòria d'accés global o compartida. Teòricament, no és imprescindible per al funcionament del programa aquest accés en forma de warps que utilitza la GPU, però sí que és molt recomanat degut al fet que l'arquitectura sempre el duu a terme d'aquesta manera, és a dir, fent un accés a 32 elements consecutius, tot i que en realitat, en aquell moment només precisi d'un d'ells.

Aquest tipus d'accés s'anomena coalescent, i el concepte gira entorn al fet que cada fil llegeixi o escrigui en una posició consecutiva en memòria, de tal manera que el nucli CUDA pugui realitzar una sola acció de 32 elements vàlids [25].

Per altra banda, també cal tenir present el fet que la GPU i la CPU no tenen el mateix espai de memòria i, per tant, que no es pot enviar o copiar les adreces des del Host al Device –és a dir, des de la CPU i la seva memòria a la GPU i la seva memòria-, ja que aquestes no es corresponen en l'una i en l'altra.

Partint de tot aquesta anàlisi, sembla doncs imperatiu adaptar el codi inicial de la xarxa neuronal escollida -on tots els accessos als elements són dispersos-, per tal d'aconseguir que les seves dades quedin consecutives en memòria.

La solució ha estat, doncs, en el manteniment de l'*array* de *structs* inicial per a poder tenir totes les capes, però on

ara cada capa contindrà 8 *arrays*, un per cadascun dels atributs que hi havia a l'*struct* de la neurona. Els *arrays* dels atributs han mantingut el nom que tenien originalment les variables relacionades, amb la única diferència que ara cada array conté el mateix atribut per totes les neurones presents a la capa. D'aquesta manera, i seguint els noms dels atributs mostrats a la Figura 2, els arrays *actv*, *bias*, *z*, *dactv*, *dbias* i *dz* tenen tantes posicions com neurones té la capa. Els arrays *out_weights* i *dw*, en canvi, tenen (num_neurones_capa_actual x num_neurones_capa_següent) posicions. Això és degut al fet que els pesos fan referència tant a la capa actual com a la següent, i és, per aquest motiu, que la última capa no precisa de cap d'aquests dos últims arrays. L'esquema de la nova distribució dels paràmetres queda, doncs, com es pot visualitzar a la Figura 3.

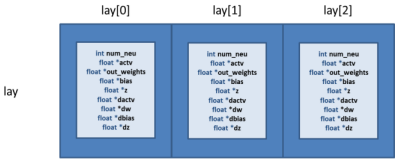

Fig. 3: Esquema actual de la xarxa neuronal

Per últim, donat que aquesta xarxa neuronal utilitza una funció d'activació per a les capes ocultes anomenada ReLU, s'ha hagut de modificar la inicialització dels pesos i de l'atribut bias per tal d'intentar incrementar al màxima la quantitat d'encerts utilitzant la inicialització Kaiming [10].

5.3 Barreja de patrons

L'entrenament del'algoritme sempre en el mateix ordre pot afectar a la fiabilitat de l'aprenentatge de la xarxa neuronal. Això pot provocar un nombre d'encerts simplement condicionat a l'ordre amb el qual s'estan obtenint els resultats i no exclusivament a l'aprenentatge de la xarxa neuronal.

Per evitar aquest problema, dins del bucle principal de la funció d'entrenament –train_neural_net(), que és l'encarregada de cridar les funcions que permetran l'entrenament de la xarxa-, i abans del bucle que recorre cadascun dels patrons, s'han creat dos bucles, la finalitat dels quals no és altra que barrejar, de manera aleatòria, tots els patrons d'entrenament de forma que mai s'entreni la xarxa en el mateix ordre -els codis relatius es poden consultar a l'apèndix A.1.1 i A.1.2-.

6 PARAL·LELITZACIÓ AMB OPENMP

Tal i com s'ha detallat en la secció 4, OpenMP utilitza programació multiprocés per tal de poder dividir una tasca i realitzar-la de forma concurrent.

Aquest apartat està centrat, doncs, en l'analisi i l'estudi de l'efecte en el rendiment de la xarxa a l'utilitzar primitives *#pragma omp for* i *#pragma omp sections*. A més a més, es busca una explicació a les causes d'aquest efecte. *#pragma omp for* permet que les iteracions d'un bucle es distribueixin i s'executin simultàneament per un equip de fils. En canvi, *#pragma omp sections* permet la creació d'un conjunt de blocs, els quals s'executaran en paral·lel per un conjunt de fils [26].

Un cop havent modificat la xarxa per a fer-la consecutiva en memòria i havent-la estudiat a fons, s'ha pogut determinar que la paral·lelització amb seccions no té cap finalitat acadèmica, ja que, en realitat, el codi és molt depenent i, exceptuant una funció, no és possible dividir-lo. És per aquest motiu que no es considera d'interés per quest treball mostrar els resultats obtinguts. En canvi, les primitives *#pragma omp for* sí que haurien d'aportar un gran canvi en el rendiment de la xarxa, ja que, donat que la majoria de bucles segueixen un patró MAP -patró que replica una mateixa operació per a tots els element d'un conjunt d'entrada [27]- el fet de repartir les iteracions entre els diferents threads (sempre que es trobin en Cores separats), agilitza l'execució del programa. Per altra banda, els patrons REDUCE -patró que combina cada element d'una col·lecció en un de sol utilitzant una funció associativa [27]- també es veuen beneficiats de la partició del codi entre els threads.

6.1 Utilitzant primitives pragma omp for

Una de les característiques que té OpenMP és la capacitat de repartir les iteracions d'un bucle entre els threads, sempre i quan les iteracions siguin independents les unes de les altres. D'aquesta manera, s'ha passat a analitzar cadascuna de les funcions utilitzades a la funció *train_neural_net* per tal de poder dividir i repartir cadascun dels bucles interns de les mateixes. El codi pertanyent a aquesta funció es pot veure en l'apèndix A.2.1.

El primer que cal fer per a poder paral·lelitzar el codi és crear una regió paral·lela. En aquest punt és molt important que aquesta no es vagi creant i destruint a cada iteració, ja que això comportaria un augment d'overhead, que és fàcilment evitable si aquesta regió es crea fora dels bucles que es paral·lelitzaran més tard.

Tanmateix, cal tenir present que el temps que s'invertirà en total serà l'equivalent al que necessita un sol thread a executar aquella part del codi, ja que, per aquesta aproximació de paral·lelització, treballarem amb threads que es trobaran en diferents Cores. Així doncs, tots els threads entraran dins de les funcions *feed_input()*, *forward_prop*, *compute_cost()*, *back_prop()* i *update_weights()*, que es poden visualitzar en l'apèndix esmentat, i serà dins dels bucles interns de cadascuna on repartiran la feina.

6.1.1 Feed_input()

Donada l'estructura de la xarxa neuronal estudiada, s'han pogut dividir la majoria de bucles, de manera que siguin fàcilment paral·lelitzables, ja que, internament, la majoria són patrons MAP.

Utilitzant la primitiva *#pragma omp for* davant dels bucles, aquests són capaços de repartir, de manera equitativa, les seves iteracions entre la quantitat de threads, tal i com es pot veure al Codi A.2.2.

6.1.2 Forward_prop()

A l'hora d'haver de decidir quin bucle paral·lelitzar, sempre s'ha de tenir en consideració la quantitat d'iteracions que està efectuant cadascun d'ells. D'aquesta manera, per norma general, es repartirà el bucle extern, ja que el treball ja queda repartit, inicialment, entre tots els threads.

Això no obstant, donat que el nombre de capes que s'està tractant en aquest treball és petit –3 en concret-, és més interessant repartir el bucle intern "j". D'aquesta manera, tots els threads faran el bucle extern –de només 2 iteracions-, però es repartiran el bucle intern, millorant així el rendiment del programa -veure codi de l'apèndix A.2.3-.

El bucle més intern es deixa sense paral·lelitzar, ja que la variable sobre la qual s'ha d'escriure ja queda repartida al dependre de "j", i la variable que depèn de "k", només es llegeix, impedint així possibles dataraces.

6.1.3 Compute_prop()

Aquesta funció -veure Codi A.2.4- mostra un patró REDUCE, al necessitar que tots els threads contribueixin a l'obtenció de la suma de la variable *tcost*. La primitiva utilitzada en aquest cas és *pragma omp for reduction(+:tcost)* indicant així que la reducció es basa en una suma contra la variable *tcost*.

Aquesta mateixa funció també mostra una part del codi que haurà de ser executada per un sol thread, degut al fet que si no el resultat no seria el correcte. Per a indicar aquesta necessitat al compilador, ens valem de la primitiva *#pragma omp single*.

Finalment, per assegurar que tots els threads se sincronitzin per a poder seguir, s'ha utilitzat una primitiva *barrier*.

6.1.4 Back_prop()

Per a la part relativa a la última capa de sortida –Output Layer-, en aquesta funció -veure Codi A.2.5- s'havia dividit el primer bucle entre tots els threads, donat que es tracta d'un patró MAP. Tanmateix, al final s'ha hagut de modificar i realitzar-lo amb un *#pragma omp single*, ja que sembla que, perquè els encerts finals siguin els mateixos, aquesta part ha de ser realitzada de manera seqüencial.

Per al segon bucle, és necessari paral·lelitzar el bucle intern "k", ja que s'escriu sobre la variable *lay[num_layers-2].dactv[k]*. De tal manera que si no es repartís aquest bucle, hi hauria problemes de dataraces. Cal dir que, donat que els límits del bucle són canviants, degut al fet que depenen del nombre de neurones de cada capa, en aquest cas no es pot aplicar un *collapse* –unió dels dos bucles per a repartir, seguidament, el conjunt de totes les iteracions d'ambdós-.

Finalment, per als bucles relatius a les capes intermitges –Hidden Layers-, donat que en aquest cas estem treballant només amb tres capes en total, s'ha decidit paral·lelitzar el bucle intern enlloc de l'extern. Aquest executa més iteracions amb diferència.

6.1.5 Update_prop()

Per últim, per a aquesta funció -veure Codi A.2.6-, degut de nou al baix nombre de capes a la xarxa, s'ha decidit paral·lelitzar els dos bucles interns "j". Pel que fa al primer, però, igual que en la funció *back_prop()*, no és possible aplicar un *collapse* dels dos bucles, ja que els límits d'ambdós són variables i dependents del nombre de neurones a cada iteració del bucle més extern.

6.2 Resultats

Un cop efectuades totes les modificacions esmentades en aquesta secció, s'ha passat a fer un estudi dels temps obtinguts amb diferentes configuracions de threads i Cores i amb diferent nombre d'iteracions per a avaluar el rendiment i la millora del programa.

Inicialment, s'ha avaluat la necessitat d'utilitzar una optimització del compilador -en aquest cas *-Ofast-* degut al fet que, realitzant només una iteració externa de manera seqüencial, el temps sense cap optimització és de 14,5590 segons, enfront d' 1,2476 segons obtinguts amb *Ofast*. La Taula 1 mostra el temps trigat i el speedup que aquest suposa, per a cada configuració de Cores i threads i per a cada conjunt d'iteracions del bucle extern. Cal esmentar també que tots aquests resultats han estat obtingut amb el cluster *Wilma* de l'Escola, així com utilitzant el mòdul de *gcc*: *gcc/10.2.0*.

Així doncs, la taula està distribuida en vuit seccions. La primera mostra el temps i els encerts obtingut per a 1, 5, 20 i 100 iteracions del bucle extern executades de manera seqüencial. La segona esbrina si, tot i que la suposició inicial d'un patró MAP ja indica que la millor configuració és d'un thread per core, el resultat obtingut és millor amb un o dos threads per Core. Un cop determinat que, efectivament, la millor configuració és la d'un thread per Core, la tercera secció mostra, amb l'opció de paral·lelització *-fopenmp*, els resultats obtingut per 4 Cores -cadascun amb un thread-. La quarta mostra el mateix procediment per a una configuració de 5 Cores. I, finalment, les restants mostren, aquest cop ja només per a una i cent iteracions, els resultats obtinguts per 6, 7, 8 i 9 Cores.

Amb els resultats obtinguts, s'observa que la millor configuració és la de 7 Cores amb un thread cadascun (la qual s'ha comprobat empíricament que és el punt d'inflexió). A mesura que es van incrementant la quantitat de Cores, el temps per cadascun dels casos també ho fa.

Això pot ser degut al fet que que l'arquitectura dels ordinadors utilitzats està formada per dos processadors de sis Cores cadascun. D'aquesta manera, mentre els threads es troben al mateix processador, a major nombre de Cores, millor és el rendiment. Tanmateix, a l'haver de canviar de processador, l'interconnexió dels threads en ambdues bandes disminueix el rendiment que oferiria l'adquisició d'un nou Core. A més a més, degut a la llei d'Amdahl, el codi, arribats a un cert punt, deixarà d'escalar, degut a la part seqüencial impossible de paral·lelitzar, de manera que la millora obtinguda ja no compensa l'overhead introduit.

7 PARAL·LELITZACIÓ AMB MPI

MPI, com ja s'ha detallat a la secció 4, utilitza un gra molt més gruixut que OpenMP. Donat que en aquest cas la paral·lelització es duu a terme mitjançant diversos processos enviats a diferentes màquines, la comunicació entre ells es fa a través de missatges, la qual cosa implica un control de la quantitat de missatges que s'envien per incrementar, en la menor mesura possible, el temps destinat a aquesta comunicació.

És necessari, llavors, modificar el codi per tal que els patrons es reparteixin entre processos -utilitzant diferents valors d'inici i fi del bucle-, i no com a OpenMP, on tot els

Optimització	nº Iteracions	nº Cores	nº Threads/core	Temps (s)	Màquina	Speedup	Encerts
Ofast	1	Seq	1	1,2476	wilma	x	721
Ofast	5	Seq	1	4,5271	wilma	x	846
Ofast	20	Seq	1	17,2742	wilma	x	898
Ofast	100	Seq	1	85,2224	wilma	x	905
Ofast	20	2	1	9,5756	wilma	1,8040	898
Ofast	20	2	2	738,6568	wilma	0,0234	898
Ofast	**1**	**4**	**1**	**0,479**	**wilma**	2,6046	721
Ofast	**5**	**4**	**1**	**1,4588**	**wilma**	3,1033	846
Ofast	**20**	**4**	**1**	**5,0298**	**wilma**	3,4344	898
Ofast	**100**	**4**	**1**	**24,7547**	**wilma**	3,4427	905
Ofast	**1**	**5**	**1**	**0,3139**	**wilma**	3,9745	721
Ofast	**5**	**5**	**1**	**1,1421**	**wilma**	3,9638	846
Ofast	**20**	**5**	**1**	**4,3121**	**wilma**	4,0060	898
Ofast	**100**	**5**	**1**	**21,2603**	**wilma**	4,0085	905
Ofast	**1**	**6**	**1**	**0,2715**	**wilma**	4,5952	721
Ofast	**100**	**6**	**1**	**19,2435**	**wilma**	4,4286	905
Ofast	**1**	**7**	**1**	**0,2592**	**wilma**	4,8133	721
Ofast	**100**	**7**	**1**	**18,4077**	**wilma**	4,6297	905
Ofast	**1**	**8**	**1**	**0,2611**	**wilma**	4,7782	721
Ofast	**100**	**8**	**1**	**18,5184**	**wilma**	4,6020	905
Ofast	**1**	**9**	**1**	**0,2747**	**wilma**	4,5417	721
Ofast	**100**	**9**	**1**	**19,3176**	**wilma**	4,4116	905

TAULA 1: OPENMP- RESULTATS I SPEEDUP OBTINGUT AMB LES DIVERSES CONFIGURACIONS DE THREADS I CORES

threads calculaven cada patró.

Pel que fa a la creació de la regió paral·lela, al contrari d'OpenMP, on s'utilitza un *#pragma omp parallel*, per a la creació dels threads, MPI no necessita crear-la com a tal de la mateixa forma. En comptes d'això, necessita una sèrie de funcions de comunicació a la funció *main*, que permeten crear la quantitat de processos dessitjada, així com establir un comunicador entre tots ells. Cada procés, per la seva banda, té un identificacor únic, que en aquest treball es recull amb el nom de la variable *my_rank* i que s'obté a partir de la funció *MPI_Comm_rank(MPI_COMM_WORLD, &my_rank)*. Per altra banda, per a obtenir el total de processos utilitzat pels càlculs es pot fer mitjançant la funció *MPI_Comm_size(MPI_COMM_WORLD, &nprocs)*, on el resultat queda emmagatzemat en la variable *nprocs*. Per últim, cal tenir present que un cop acaba el programa, cal executar la funció *MPI_Finalize()*. L'esquema d'aquestes crides es pot consultar en l'apèndix A.3.1

A l'hora de repartir les iteracions entre els processos, s'ha de tenir present que aquesta repartició no sempre serà completament equitativa, ja que la divisió entre processos i iteracions no sempre serà exacta. Així doncs, cal triar un criteri a seguir en el cas d'iteracions sobrants –repartició equitativa, assignació totes les iteracions sobrants a un procés concret, etc.-.

En el cas d'aquest treball, s'ha escollit la primera opció, donat que és la que acaba balancejant millor la càrrega de cada procés. El codi que queda després d'aquesta repartició, així com la seva explicació, poden ser consultats a l'apèndix A.3.2.

7.1 Comunicació utilitzant col·lectives

El fet de repartir els patrons d'entrenament entre els processos té els seus avantatges i inconvenients. Per una banda, el treball es reparteix, reduint el temps d'execució de la xarxa. Tanmateix, per una altra, cada procés pot acabar amb uns pesos lleugerament diferents, depenent dels valors que hagi aplicat als seus patrons. És per aquest motiu que, un cop acabada cada iteració del bucle extern i abans de tornar a executar la següent, és necessari arribar a un resultat conjunt pel que fa als pesos de cadascuna de les neurones de cada capa. Aquesta secció se centra en l'utilització de la col·lectiva *MPI_Allreduce()* per aconseguir aquest objectiu.

MPI_Allreduce combina els valors de tots els processos per, seguidament, distribuir de nou el valor resultant a cadascun d'ells. D'aquesta manera, tots els processos tenen la suma global dels pesos obtinguts entre tots ells.

A continuació, i un cop ja cada procés conté la suma de tots els pesos, es divideix el resultat d'aquesta suma entre tots els processos per optenir una mitjana, la qual cosa resulta molt pràctica en termes de còmput. D'aquesta manera, tots els processos es queden amb el valor mitjà calculat a la vegada, sense haver d'esperar que un dels processos rebi tots els pesos, hagi de fer la mitjana entre tots els valors -mentre els altres processos s'esperen- i, finalment, enviï de nou el resultat a cadascun dels processos per a la següent iteració.

La primitiva *MPI_Allreduce* utilitza diferents paràmetres: una opció per indicar que tant el buffer emissor com el receptor són el mateix –MPI_IN_PLACE-, l'especificació de quin atribut es vol compartir entre tots els processos –en aquest cas el out_weights de cada capa-, la mida de l'array a compartir –referit només a la mida de cadascun dels processos, no del conjunt global-, una especificació del tipus de dades amb les quals treballarem –en aquest cas de tipus float-, l'especificació de quina operació es vol fer al REDUCE –en aquest cas una suma- i, finalment, el comunicador que utilitzaran tots els processos. El codi vinculat a aquesta explicació està disponible a l'apèndix A.3.3

7.2 Comunicació punt a punt

Pel que fa a la comunicació punt a punt, es fa transmetent dades entre dos terminals, en aquest cas dos processos, per un canal únic.

En aquest cas, la idea gira entorn al fet que, un cop ha acabat la primera iteració externa i abans de la següent, cada procés enviï el seu valor del *out_weights* al procés amb identificador zero. A partir d'aquí, aquest és l'encarregat de sumar tots els valors dels pesos -la qual cosa la primitiva *MPI_Allreduce()* feia directament-. Un cop obtinguts i sumats tots els valors, és el procés zero també l'encarregat de fer la mitjana entre tots ells i enviar el resultat de nou a cadascun dels processos perquè puguin continuar amb la següent iteració de la xarxa neuronal.

Així doncs, la comunicació punt a punt, en comparació amb la de col·lectiva de la subsecció anterior, té com avantatge que hi ha menys comunicació entre processos, ja que no cal que tots enviïn el resultat els uns als altres, sinó que només ho fa el procés zero. Com a contrapartida, mentre aquest està efectuant els càlculs, la resta ha d'esperar fins a obtenir el resultat. El codi vinculat a aquesta explicació està disponible a l'apèndix A.3.4

7.3 Resultats

Els resultats obtinguts per les seccions 7.1 i 7.2 es recullen en les taules adjuntes a l'apèndix A.3.5.

Cal indicar que, a diferència d'OpenMP, en aquest cas s'ha estudiat el rendiment de la xarxa utilitzant sempre 100 iteracions del bucle extern i una optimització *Ofast*, modificant el nombre de tasques -processos-, entre les quals es reparteixen els patrons. A més a més, a cada node es poden executar fins a 12 tasques.

MPI no proporciona mai el mateix resultat que OpenMP, ja que cada procés treballa amb diferents patrons i no com a

OpenMP, on els threads processaven, en paral·lel, el mateix patró.

Analitzant les taules, s'observa com la primitiva *MPI_Allreduce* proporciona un millor resultat que les punt a punt, degut al fet que, tot i que hi ha més comunicació entre processos, aquests no s'han d'esperar a la finalització dels càlculs del procés zero, així com a l'enviament d'aquests a la resta de processos. A més a més, es constata com el codi és completament paral·lelitzable, ja que el temps d'execució es redueix a mesura que augmenten el nombre de tasques que l'executen, és a dir, el nombre de processos.

Amb les primitives punt a punt -*Send()* i *Recv()*-, tot i que el resultat no sigui tan òptim, també se segueix, aproximadament, el mateix patró de rendiment.

Ambdós rendiments poden ser visualitzats en la gràfica de la Figura 4, on l'eix d'abscisses representa els processos utilitzats i l'eix d'ordenades el temps (en segons) que triga la xarxa en executar-se.

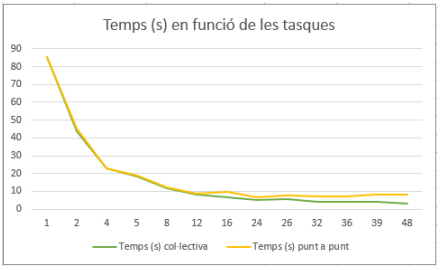

Fig. 4: Rendiment obtingut amb MPI_Allreduce i punt a punt

8 CUDA

CUDA aplica paral·lelisme permetent definir funcions anomenades *kernels*, les quals es defineixen utilitzant l'especificador de declaració *__global__*. Aquestes funcions s'executen N vegades en paral·lel per a M fils CUDA, els quals són cridats des del Host mitjançant la sintaxi

$$<<< blocs, threads >>>$$

Aquesta sintaxi declararà entre quants blocs volem distribuir l'estructura de dades que enviem i entre quants threads per bloc la voldrem repartir. Així doncs, cada fil que executa el kernel rep un identificador de fil únic, al qual es pot accedir utilitzant la crida *threadIdx.x* –en el cas de voler només una dimensió- i que serà accessible dins del nucli. Serà així com s'iterarà cadascun dels elements de l'estructura de dades enviada al Device, és a dir, a la GPU. Cal esmentar, però, que el component *threadIdx* és un vector de tres components, les quals poden ser identificades utilitzant un índex d'una, dues o tres dimensions [28].

Pel que fa a les estructures de dades a enviar, es crearan utilitzant les funcions *cudaMalloc* i *cudaMemcpy*, ja que és necessari al·locatar i copiar la informació del Host al Device per a poder realitzar els càlculs. Cal tenir present el fet que CUDA treballa amb la GPU, i aquesta i la CPU no tenen el mateix espai de memòria. Per tant, no es pot enviar o copiar des del Host al Device les adreces, ja que aquestes no es corresponen en una i en l'altra. És per aquest motiu que cada cop que el Device vulgui utilitzar les dades calculades al Host o viceversa, caldrà utilitzar la funció *cudaMemcpy*. El Device, però, pot reutilitzar, entre kernel i kernel, els valors de les estructures que ja té guardats. A més a més, cal recordar que, tal i com s'ha explicat ja a la secció 5.2, que les dades s'hauran de trobar consecutives en memoria.

8.1 Adaptació d'estructures

Tenint en compte tots aquests conceptes, per tal de poder paral·lelitzar la xarxa neuronal amb CUDA i, consegüentment, enviar els arrays dels atributs de cada capa a la GPU, s'ha dissenyat un sol array, anomenat *d_lay*, el qual conté, de manera consecutiva, tots els atributs per a cadascuna de les capes, tal i com mostra la Figura 5. Així doncs, per a al·locatar l'espai de memòra necessari i mostrar una codificació relativament senzilla a nivell educatiu, s'ha acumulat, en una variable, la mida de cadascuna de les capes, de manera que, seguint una metodologia repetitiva, s'obtingui la mida necessària per aquest nou array *d_lay*.

A més a més, cal recordar que no tots els atributs de cada capa tenen el mateix espai de memòria, ja que els pesos guarden una quantitat major de neurones.

Fig. 5: Estructura de memòria *d_lay*

A l'apèndix A.4.1, es poden observar les tres estructures extra creades per a poder treballar amb tots els elements necessaris al Device. Aquestes són els apuntadors *d_lay*, *d_input* i *d_desired_outputs*. Un cop declarats, cal al·locatar la memòria necessària per cadascun utilitzant la funció *cudaMalloc()*, tal i com es pot consultar a l'apèndix A.4.2. En el cas de l'array *d_lay*, com s'ha esmentat, caldrà utilitzar la variable amb l'acumulació de la mida de cadascuna de les capes.

Per acabar amb la creació de les estructures, cal utilitzar la funció *cudaMemcpy()*, per tal de poder copiar totes les estructures del Host al Device i viceversa, de manera que aquest pugui treballar amb cadascuna de les dades als kernels. En aquest sentit, cal indicar que a CUDA s'intenten minimitzar al màxim les copies de memòria, ja que són les que suposen un major temps d'execució. És per aquest motiu que s'ha seguit la idea ja esmentada per a OpenMP, on la regió paral·lela es creava fora del bucle més extern per tal de no crear-la i destruir-la per a cada iteració -la qual cosa hagués suposat un increment del temps d'execució-. Així doncs, les copies del Host al Device també es realitzen abans d'entrar al bucle més extern. De la mateixa manera, les copies del Device al Host també es realitzen un cop acabat el bucle.

Tenint en compte que tota l'estructura de dades ara es troba en un sol *array*, és necessari utilitzar indicadors que permetin accedir a l'atribut desitjat de cada capa creant, d'aquesta manera, els *arrays displacement*, *access* i *access_weights*. Aquests, permeten, respectivament, posicionar-se sobre la capa on es vol treballar, accedir a l'atribut a modificar -sense comptar els destinats als pesos- i, ara sí, actuar sobre aquests últims. El codi complert referit a aquesta explicació pot ser consultat a l'apèndix A.4.3.

8.2 Kernels

Pel que fa a la paral·lelització de cadascuna de les funcions, CUDA no utilitza cap primitiva, com era el cas d'OpenMP o es distribueixen els patrons, com feia MPI. En aquest cas, l'estructura de dades creada s'envia a la GPU utilitzant l'estructura de bloc i thread explicada a l'inici d'aquesta secció de la següent manera:

$$nom\_kernel <<< blocs, threads >>> (paràmetres)$$

Així doncs, els paràmetres generals a enviar per a cadascuna de les funcions són l'estructura *d_lay*, la capa sobre la qual cal actuar, el valor del desplaçament que cal realitzar per passar a les capes posteriors i les variables d'accés *access* i *access_weights* per a poder accedir a l'atribut sobre el qual es vol operar.

Dins de cadascun dels kernels, els blocs i els threads indicats durant la crida, es recullen sobre una variable, que tindrà un identificador únic per a cadascun d'ells, de manera que es pugui simular el comportament d'un bucle *for* de manera paral·lela. D'aquesta forma, utilitzant una sola operació condicionada a aquest identificatiu, cada thread de cada block és capaç de realitzar l'operació i emmagantzemar el resultat sobre la posició de destinació, és a dir, que cada thread de cada bloc és capaç d'accedir a una posició de l'array simultàneament, permetent així que totes les posicions de l'equivalent bucle *for* es tractin a la vegada.

Pel que fa a les estructures condicionants, una estructura *if-else* a CUDA seria molt ineficient, degut al fet que aquesta trencaria el paral·lelisme exposat anteriorment. És per aquest motiu que, en aquest cas, cal utilitzar operadors ternaris -veure apèndix A.4.5 i A.4.8-, permetent així realitzar totes les condicions a la vegada. Per últim, pel que fa als sumatoris contra una variable determinada -veure apèndix A.4.5- CUDA requereix de la necessitat de la creació d'una variable externa, de tantes posicions con iteracions tindria el bucle *for* corresponent, i que contingui tots els resultats de l'operació designada per a cadascuna de les seves posicions. A continuació, realitza una reducció -fent servir un algorisme optimitzat en forma d'arbre i complexitat logarítmica- sobre la variable, de tal manera que, finalment, obté el resultat global del sumatori a la seva posició zero. Al realitzar aquesta operació de reducció, cal saber, però, que en el cas que el nombre de posicions d'aquest array auxiliar no sigui una potència de 2 -donat que la inicialització del bucle sumatori s'inicia amb la divisió entre 2 de la mida- cal calcular el següent valor més proper al valor real que sigui potència de 2, i, dins del bucle, contemplar que no realitzi la reducció per valors majors a la mida real de l'array.

La codificació de la resta de kernels pot ser consultada als apèndix A.4.4, A.4.6, A.4.7 i A.4.9.

8.3 Resultats

Els resultats obtinguts per a la secció 8 es recullen a la Taula 2. Primer es mostren els resultat obtinguts sense cap paral·lelització de la xarxa i executats als ordinadors aolin -utilitzant una GPU GeForceRTX3080- per 1, 4, 20, 100, 200 i 500 iteracions. Seguidament, s'ha realitzat el mateix procediment però, aquest cop, executant el codi amb les transformacions ja pertinents per a CUDA.

Optimització	nº Iteracions	Tipus	Encerts	Temps (s)	Màquina	Speedup
Ofast	1	Seq	721	0,7289	aolin	x
Ofast	5	Seq	846	3,3073	aolin	x
Ofast	20	Seq	898	11,4295	aolin	x
Ofast	100	Seq	905	56,7937	aolin	x
Ofast	200	Seq	909	113,5439	aolin	x
Ofast	500	Seq	909	283,3323	aolin	x
Ofast	**1**	**CUDA**	**721**	**0,8900**	**aolin**	0,8190
Ofast	**5**	**CUDA**	**869**	**1,1400**	**aolin**	2,9011
Ofast	**20**	**CUDA**	**889**	**1,9900**	**aolin**	5,7435
Ofast	**100**	**CUDA**	**906**	**7,0900**	**aolin**	8,0104
Ofast	**200**	**CUDA**	**905**	**12,7800**	**aolin**	8,8845
Ofast	**500**	**CUDA**	**913**	**30,3800**	**aolin**	9,3263

TAULA 2: RESULTATS DE RENDIMENT AMB CUDA

És en aquest aspecte on realment es nota l'impacte que té el fet de realitzar només un cop les copies a memòria del Device al Host i viceversa. Observant la Taula 2, s'aprecia com el speedup del codi, és a dir, la seva millora, respecte al temps d'execució, augmenta a mesura que el nombre d'iteracions també ho fa. Donat que la copia a memòria s'haurà de dur a terme, tant si es realitza només una iteració com si es realitzen 500, l'impacte de la paral·lelització aplicada al codi a l'utilitzar els diversos kernels, serà més identificable a mesura que la millora del rendiment -deguda a la paral·lelització- sigui major que el temps destinat a fer totes les copies dels atributs de cada capa.

9 CONCLUSIONS

Un cop obtinguts tots els resultats, es pot determinar que, per aquesta xarxa neuronal en qüestió, en el cas de voler-se executar un nombre alt d'iteracions -considerant com a alt un nombre major o igual a 100-, les aproximacions que donen un rendiment millor són les d'MPI i CUDA. Això ve condicionat per la manera com estan estructurats els Cores en els ordinadors de l'Escola, ja que, per a OpenMP, el rendiment es veu disminuit en el moment que necessita intercomunicar els threads entre els dos processadors, cadascun dels quals compta amb sis Cores.

D'aquesta manera, mentre el nombre de patrons d'entrenament no augmenti de 1934 i les iteracions del bucle extern siguin properes a 100, MPI i CUDA proporcionen un rendiment aproximat, tot i que si s'utilitzen col·lectives, el rendiment amb MPI és lleugerament millor. Això no obstant, en el cas de voler realitzar moltes més iteracions del bucle extern, el rendiment d'MPI respecte a CUDA començarà a decaure, degut al fet que el temps destinat a l'intercanvi de missatges en el primer s'incrementa segons el nombre d'iteracions. A OpenMP o a CUDA, en canvi, donat que la regió paral·lela o les còpies entre el Host i el Device es realitzen abans dels bucles, el fet d'incrementar el nombre d'iteracions augmenta el rendiment de la xarxa, ja que a més iteracions, la millora de la paral·lelització, respecte a l'increment de temps que suposa crear aquest paral·lelisme, és major.

De cara a un treball futur, quedaria pendent, per exemple, refactoritzar el codi en diferents fitxers, així com provar el rendiment i l'entrenament de la xarxa utilitzant diferentes configuracions de capes i neurones.

AGRAÏMENTS

Aquest treball no hauria estat possible sense l'ajut, el suport i la dedicació de la doctora Anna Bàrbara Sikora, tutora d'a-

quest treball, i del doctor Eduardo Cesar Galobardes.

REFERÈNCIES

[1] "Qué son las redes neuronales y sus funciones," https://www.atriainnovation.com/que-son-las- redes-neuronales-y-sus-funciones/, consultat el: 2021-09-26.

[2] J. GONZÁLEZ RAMÍREZ, "Qué es y que aplicaciones tiene una red neuronal artificial," https://www.datacentric.es/blog/insight/red-neuronal-artificial-aplicaciones/, consultat el: 2021-09-26.

[3] "Diferència entre aprenentatge automàtic i intel·ligència artificial," https://ca.living-in-belgium.com/difference-between-machine-learning-and-artificial-intelligence-148, 2022, consultat el: 2022-01-29.

[4] O. SIMEONE, "A very brief introduction to machine learning with applications to communication systems," https://arxiv.org/pdf/1808.02342.pdf, 2018, consultat el: 2021-09-29.

[5] Y. H. G. LECUN, Yann; BENGIO, "Deep learning," https://www.nature.com/articles/nature14539, 2015, consultat el: 2021-09-29.

[6] "Neural networks," https://www.ibm.com/docs/es/spss-modeler/SaaS?topic=networks-neural-model, consultat el: 2021-09-21.

[7] "Introducción a las redes neuronales pt. i," https://futurelab.mx/redes%20neuronales/inteligencia%20artificial/2019/06/25/intro-a-redes-neuronales-pt-1/, consultat el: 2022-01-30.

[8] "Redes neuronales," https://disi.unal.edu.co/ lctorress/RedNeu/LiRna008.pdf, consultat el: 2022-01-30.

[9] "El modelo de redes neuronales," https://www.ibm.com/cloud/learn/neural-networks, consultat el: 2021-09-21.

[10] M. MARVIN WANG, "Rectified linear unit (relu) and kaiming initialization," https://medium.com/ai%C2%B3-theory-practice-business/the-rectified-linear-unit-relu-and-kaiming-initialization-4c7a981dfd21, consultat el: 2021-11-02.

[11] I. ALVARADO, "Redes neuronales," https://ml4a.github.io/ml4a/es/neural_networks/, consultat el: 2021-09-23.

[12] "El perceptrón," https://sierterm.es/content/perceptr%C3%B3n/, consultat el: 2021-09-24.

[13] "¿qué es la hpc?" https://www.intel.es/content/www/es/es/high-performance-computing/what-is-hpc.html, consultat el: 2021-09-24.

[14] R. M. P. FOX, Geoffrey; WILLIAMS, *Parallel Computing Works!* Morgan Kaufmann, 1994.

[15] "Capítulo 2: Técnicas de paralelización automática," https://www.tdx.cat/bitstream/handle/10803/5983/08Eap08de14.pdf?sequence=8isAllowed=y, consultat el: 2021-09-24.

[16] G. AMDAHL, "Validity of the single processor approach to achieving large-scale computing capabilities," pp. 483–485, 1967.

[17] "Openmp," https://www.openmp.org/, consultat el: 2021-09-25.

[18] "Message passing interface," https://www.mpi-forum.org/docs/, consultat el: 2021-09-25.

[19] "Cuda zone," https://developer.nvidia.com/cuda-zone, consultat el: 2021-09-25.

[20] S. BECERRA, "Simple neural network implementation in c," https://towardsdatascience.com/simple-neural-network-implementation-in-c-663f51447547, 2019, consultat el: 2021-09-16.

[21] "Simple neural network in c," https://codereview.stackexchange.com/questions/191498/simple-neural-network-in-c, consultat el: 2021-09-17.

[22] "Ml – neural network implementation in c++ from scratch," https://www.geeksforgeeks.org/ml-neural-network-implementation-in-c-from-scratch/, consultat el: 2021-09-17.

[23] M. BHOLE, "Neural-network-framework-using-backpropogation-in-c," https://github.com/mayurbhole/Neural-Network-framework-using-Backpropogation-in-C, consultat el: 2021-09-19.

[24] O. A. F. A. M. A. J. I. M. R. J. ARBELAITZ GALLEGO, Olatz; ARREGI URIARTE, "Programación paralela: Mpi," https://addi.ehu.es/bitstream/handle/10810/20501/martin_aramaburu-12-2016-ik.pdf?sequence=1, p.27 Consultat el: 2021-11-01.

[25] D. BURGOS DOMÍNGUEZ, "Aplicación de las estrategias de paralelización de algoritmos," https://uvadoc.uva.es/bitstream/handle/10324/20925/TFG-G2240.pdf?sequence=1isAllowed=y, p.68 Consultat el: 2021-11-01.

[26] U. de Granada, "Programación paralela," https://lsi2.ugr.es/jmantas/ppr/ayuda/omp_ayuda.php?ayuda=omp_directivas, consultat el: 2022-01-17.

[27] C. G. J. G. L. M. C. P. S. T. Y. O. J. N. J. C. D. S. S. S. A. C. D. P. J. H. P. B. C. BERNAL, Fabián; ALBARRACÍN, "Programación paralela," http://ferestrepoca.github.io/paradigmas-de-programacion/paralela/paralela_teoria/index.html, consultat el: 2022-01-16.

[28] "Cuda c++ programming guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, consultat el: 2021-12-17.

APÈNDIX

A.1 Modificacions del codi inicial

A.1.1 Codi per barrejar els patrons

Apèndix vinculat a la Figura 6.

```
for(int p = 0; p < NUMPAT; p++)
{
        ranpat[p] = p;
}
for(int p = 0; p < NUMPAT; p++)
{
        int x = rando();
        int np = (x * x) % NUMPAT;
        int op = ranpat[p]; ranpat[p] = ranpat[np]; ranpat[np] = op;
}
```

Fig. 6: Codi per barrejar els patrons

A.1.2 Funció rando()

Apèndix vinculat a la Figura 7.

```
int rando()
{
        seed = (214013*seed+2531011);
        return seed >>16;
}
```

Fig. 7: Funció rando()

A.2 OpenMP

A.2.1 Funció train_neural_net()

Apèndix vinculat a la Figura 8.

```
void train_neural_net(void)
{
    int ranpat[NUMPAT];

    #pragma omp parallel
    {
        for(int it=0;it<100;it++)
        {
                #pragma omp for
                for(int p = 0; p < NUMPAT; p++)
                {
                        ranpat[p] = p;
                }
                #pragma omp single
                {
                        for(int p = 0; p < NUMPAT; p++)
                        {
                           int x = rando();
                           int np = (x * x) % NUMPAT;
                           int op = ranpat[p]; ranpat[p] =
ranpat[np]; ranpat[np] = op;
                        }
                }
                for(int i=0;i<num_training_ex;i++)
                {
                        int p = ranpat[i];
                        feed_input(p);
                        forward_prop();
                        compute_cost(p);
                        back_prop(p);
                        update_weights();
                }
        }
    }
}
```

Fig. 8: Funció train_neural_net()

A.2.2 Funció feed_input()

Apèndix vinculat a la Figura 9.

A.2.3 Funció forward_prop()

Apèndix vinculat a les figures 10 i 11.

```
void feed_input(int i)
{
        #pragma omp for
        for(int j=0;j<num_neurons[0];j++)
        {
                lay[0].actv[j] = input[i][j];
        }
}
```

Fig. 9: Funció feed_input()

```
2.  void forward_prop()
3.  {
4.      for(int i=1;i<num_layers;i++)
5.      {
6.        #pragma omp for
7.        for(int j=0;j<num_neurons[i];j++)
8.        {
9.          lay[i].z[j] = lay[i].bias[j];
10.         for(int k=0;k<num_neurons[i-1];k++)
11.         {
              lay[i].z[j] += ((lay[i-1].out_weights[j*num_neurons[i-1] + k])*
    (lay[i-1].actv[k]));
12.         }
13.         // Relu Activation Function for Hidden Layers
14.         if(i< num_layers-1)
15.         {
16.           if((lay[i].z[j]) < 0)
17.           {
18.             lay[i].actv[j] = 0;
```

Fig. 10: Funció forward_prop() (primera part)

```
19.           }
20.           else
21.           {
22.             lay[i].actv[j] = lay[i].z[j];
23.           }
24.         }
25.         else
26.         {
27.           lay[i].actv[j] = 1/(1+exp(-lay[i].z[j]));
28.
29.         }
30.       }
31.     }
}
```

Fig. 11: Funció forward_prop() (segona part)

A.2.4 Funció compute_cost()

Apèndix vinculat a la Figura 12.

```
1.   void compute_cost(int i)
2.   {
3.       float tmpcost=0;
4.       #pragma omp for reduction(+:tcost)
5.       for(int j=0;j<num_neurons[num_layers-1];j++)
6.       {
7.           tmpcost = desired_outputs[i][j] - lay[num_layers-1].actv[j];
8.           cost[j] = (tmpcost * tmpcost)/2;
9.           tcost+= cost[j];
10.      }
11.      #pragma omp single
12.      {
13.              full_cost = (full_cost + tcost)/n;
14.              n++;
15.      }
16.      #pragma omp barrier
17.  }
```

Fig. 12: Funció compute_cost()

A.2.5 Funció back_prop()

Apèndix vinculat a la Figura 13.

A.2.6 Funció update_weights()

Apèndix vinculat a la Figura 14.

A.3 MPI

A.3.1 Inicialització dels processos

Apèndix vinculat a la Figura 15.

A.3.2 Repartició de les iteracions del bucle

El codi de la Figura 16 comença creant unes variables de tipus *integer -extrai*, *extrafi*, *ini* i *fi-* que permetran establir els límits del bucle per cadascun dels processos.

```
void back_prop(int p)
{
    // Output Layer
    #pragma omp single
    for(int j=0;j<num_neurons[num_layers-1];j++)
    {
      lay[num_layers-1].dz[j] = (lay[num_layers-1].actv[j] -
    desired_outputs[p][j]) * (lay[num_layers-1].actv[j]) * (1-
    lay[num_layers-1].actv[j]);

      lay[num_layers-1].dbias[j] = lay[num_layers-1].dz[j];
    }
    #pragma omp barrier
    for(int j=0;j<num_neurons[num_layers-1];j++)
    {
      #pragma omp for
      for(int k=0;k<num_neurons[num_layers-2];k++)
      {
         lay[num_layers-2].dw[j*num_neurons[num_layers-2] + k] =
    (lay[num_layers-1].dz[j] * lay[num_layers-2].actv[k]);

         lay[num_layers-2].dactv[k] = lay[num_layers-
    2].out_weights[j*num_neurons[num_layers-2] + k] * lay[num_layers-
    1].dz[j];
      }
    }
    // Hidden Layers
    for(int i=num_layers-2;i>0;i--)
    {
      #pragma omp for
      for(int j=0;j<num_neurons[i];j++)
      {
         if(lay[i].z[j] >= 0)
         {
               lay[i].dz[j] = lay[i].dactv[j];
         }
         else
         {
               lay[i].dz[j] = 0;
         }

         for(int k=0;k<num_neurons[i-1];k++)
         {
            lay[i-1].dw[j*num_neurons[i-1] + k] = lay[i].dz[j] * lay[i-
    1].actv[k];

            if(i>1)
            {
               lay[i-1].dactv[k] = lay[i-1].out_weights[j* num_neurons[i-1]
    + k] * lay[i].dz[j];
            }
         }

        lay[i].dbias[j] = lay[i].dz[j];
       }
    }
}
```

Fig. 13: Funció back_prop()

```
1.  void update_weights(void)
2.  {
3.      for(int i=0;i<num_layers-1;i++)
4.       {
5.          #pragma omp for
6.          for(int j=0;j<num_neurons[i+1];j++)
7.          {
8.             for(int k=0;k<num_neurons[i];k++)
9.             {
10.               // Update Weights
11.               lay[i].out_weights[j*num_neurons[i]+ k] =
    (lay[i].out_weights[j*num_neurons[i]+ k]) - (alpha *
    lay[i].dw[j*num_neurons[i]+ k]);
12.
13.            }
14.         }
15.         #pragma omp for
16.         for(int j=0;j<num_neurons[i];j++)
17.         {
18.            // Update Bias
19.            lay[i].bias[j] = lay[i].bias[j] - (alpha * lay[i].dbias[j]);
20.         }
21.      }
22. }
```

Fig. 14: Funció update_weights()

```
int main(int argc, char *argv[])
{
    int i;
    clock_t start = clock();
    int my_rank, nprocs;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size( MPI_COMM_WORLD, &nprocs);

    //CODI

    MPI_Finalize();
    clock_t end = clock();
    return 0;
}
```

Fig. 15: Funció main

A partir d'aquí, el primer pas consisteix a repartir les iteracions sobrants de manera equitativa entre els processos. A mode d'exemple i per a facilitar la seva comprensió, suposant que tenim 24 processos d'entrada, inicialment tots realitzaran 1934 / 24 = 80 iteracions, sobrant-n'hi 14. Així doncs, per tal de repartir-ho de manera balancejada, els processos del 0 al 9 continuaran realitzarant el mateix nombre d'iteracions i no serà fins al procés 24 - (1934 - (24 * (1934/24))) = 10 que s'anirà afegint una iteració a cadascun dels restants per a fer les 14 que falten. Les variables *extrai* i *extrafi* s'aniran actualitzant per a permetre, finalment, obtenir els límits del bucle per a cada procés, delimitats per les variables *ini* i *fi*.

En el suposat cas que es creessin més processos que iteracions a repartir, a la inicialització es regula el fet que no es reparteixi feina als processos sobrants.

```
void train_neural_net(int my_rank, int nprocs)
{
    int ranpat[NUMPAT];
    int extrai = 0, extrafi = 0, ini = 0, fi = 0;

    //Repartir iteracions sobrants de manera equitativa
    for(int i = nprocs - (num_training_ex - (nprocs *
(num_training_ex/nprocs))); i <= my_rank; i++)
    {
       if(my_rank == nprocs-(num_training_ex % nprocs))
       {
              extrai = 0;
              extrafi = 1;
       }
       else
       {
              extrai++;
              extrafi++;
       }
    }
    //Inicialització
    if(nprocs > num_training_ex)
    {
       if(my_rank < num_training_ex)
       {
          ini = my_rank;
          fi = my_rank + 1;
       }
       else
       {
          ini = 0;
          fi = 0;
       }
    }
    else
    {
       ini = my_rank * (num_training_ex/nprocs) + extrai;
       fi = (my_rank + 1) * (num_training_ex/nprocs) + extrafi;
    }
    for(int it=0;it<20;it++)
    {
       //BARREJA DE PATRONS
       for(int i=ini;i<fi;i++)
       {
              int p = ranpat[i];

              feed_input(p);
              forward_prop();
              compute_cost(p);
              back_prop(p);
              update_weights();
       }
    }
}
```

Fig. 16: Repartició de les iteracions del bucle

A.3.3 Utilització de col·lectives

Apèndix vinculat a la Figura 17.

```
for(int i=0;i<num_layers-1;i++)
{
       MPI_Allreduce(MPI_IN_PLACE, lay[i].out_weights,
num_neurons[i+1]*num_neurons[i], MPI_FLOAT, MPI_SUM,
MPI_COMM_WORLD);

       for(int j=0;j<num_neurons[i+1];j++)
       {
              for(int k=0;k<num_neurons[i];k++)
              {
                     // Update Weights
                     lay[i].out_weights[j*num_neurons[i] + k] /=
nprocs;
              }
       }
}
```

Fig. 17: Primitiva col·lectiva MPI_Allreduce()

Col·lectives

Optimització	nº Iteracions	nº Nodes	nº Tasks	Temps (s)	Màquina	Speedup	Encerts
Ofast	100	Seq	1	85,2224	wilma	x	905
Ofast	**100**	**1**	**2**	**43,9309**	**wilma**	1,9399	915
Ofast	**100**	**1**	**4**	**22,9582**	**wilma**	3,7121	910
Ofast	**100**	**1**	**5**	**18,1912**	**wilma**	4,6848	906
Ofast	**100**	**1**	**8**	**11,5990**	**wilma**	7,3474	906
Ofast	**100**	**1**	**12**	**8,3745**	**wilma**	10,1764	900
Ofast	**100**	**2**	**16**	**6,8184**	**wilma**	12,4989	899
Ofast	**100**	**2**	**24**	**5,1960**	**wilma**	16,4015	893
Ofast	**100**	**3**	**26**	**5,3966**	**wilma**	15,7919	891
Ofast	**100**	**3**	**32**	**4,0947**	**wilma**	20,8129	891
Ofast	**100**	**3**	**36**	**3,9985**	**wilma**	21,3136	885
Ofast	**100**	**4**	**39**	**4,0208**	**wilma**	21,1954	883
Ofast	**100**	4	**48**	**3,2259**	**wilma**	26,4182	878

TAULA 3: MPI-RESULTATS I SPEEDUP OBTINGUT AMB LES DIVERSES CONFIGURACIONS PER MPI_ALLREDUCE

Punt a punt

Optimització	nº Iteracions	nº Nodes	nº Tasks	Temps (s)	Màquina	Speedup	Encerts
Ofast	100	Seq	1	85,2224	wilma	x	905
Ofast	**100**	**1**	**2**	**45,1948**	**wilma**	1,8857	915
Ofast	**100**	**1**	**4**	**22,9770**	**wilma**	3,7090	915
Ofast	**100**	**1**	**5**	**18,9174**	**wilma**	4,5050	905
Ofast	**100**	**1**	**8**	**12,1308**	**wilma**	7,0253	907
Ofast	**100**	**1**	**12**	**8,8267**	**wilma**	9,6551	903
Ofast	**100**	**2**	**16**	**8,41**	**wilma**	10,1335	898
Ofast	**100**	**2**	**24**	**6,6105**	**wilma**	12,8920	892
Ofast	**100**	**3**	**26**	**7,6900**	**wilma**	11,0822	887
Ofast	**100**	**3**	**32**	**6,9014**	**wilma**	12,3486	891
Ofast	**100**	**3**	**36**	**7,0462**	**wilma**	12,0948	885
Ofast	**100**	**4**	**39**	**8,2599**	**wilma**	10,3176	883
Ofast	**100**	4	**48**	**7,9525**	**wilma**	10,7164	877

TAULA 4: MPI-RESULTATS I SPEEDUP OBTINGUT AMB LES DIVERSES CONFIGURACIONS PER PUNT A PUNT

A.3.4 Utilització de primitives punt a punt

Apèndix vinculat a la Figura 18.

```
for(int i=0;i<num_layers-1;i++)
{
  if(my_rank != 0)
  {
     MPI_Send(lay[i].out_weights, num_neurons[i+1]*num_neurons[i],
MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
     MPI_Recv(lay[i].out_weights, num_neurons[i+1]*num_neurons[i],
MPI_FLOAT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }
  else
  {
     float *aux;
     aux = (float*)malloc(num_neurons[i+1]*num_neurons[i]*sizeof(float));

     for(int l = 1; l<nprocs; l++)
     {
        MPI_Recv(aux, num_neurons[i+1]*num_neurons[i], MPI_FLOAT,
l, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        for(int j=0;j<num_neurons[i+1];j++)
        {
          for(int k=0;k<num_neurons[i];k++)
          {
              // Update Weights
              lay[i].out_weights[j*num_neurons[i] + k] +=
aux[j*num_neurons[i] + k];
          }
        }
     }

     for(int j=0;j<num_neurons[i+1];j++)
     {
        for(int k=0;k<num_neurons[i];k++)
        {
           // Update Weights
           lay[i].out_weights[j*num_neurons[i] + k] /= nprocs;
        }
     }
     for(int l = 1; l<nprocs; l++)
     {
        MPI_Send(lay[i].out_weights,
num_neurons[i+1]*num_neurons[i], MPI_FLOAT, l, 0, MPI_COMM_WORLD);
     }
  }
}
```

Fig. 18: Primitives punt a punt Send() i Recv()

A.3.5 Resultats MPI

Apèndix vinculat a les taules 3 i 4.

A.4 CUDA

A.4.1 Estructures al Device

Apèndix vinculat a la Figura 19.

```
float *d_lay;
char *d_input;
float *d_desired_outputs;
```

Fig. 19: Estructures utilitzades per a passar les dades del Host al Device

A.4.2 cudaMalloc()

Apèndix vinculat a la Figura 20.

```
//d_input
      cudaMalloc((void**)&d_input, NUMPAT*NUMIN*sizeof(char));

//d_desired_outputs
  cudaMalloc((void**)&d_desired_outputs, NUMPAT*NUMOUT*sizeof(float));

//d_lay
  int totallay = 0;
  for(int i = 0; i < num_layers; i++)
  {
     if( i < num_layers - 1)
     {
       totallay += (6 * num_neurons[i] + 2 * num_neurons[i+1] *
num_neurons[i]);
     }
     else
     {
       totallay += (6 * num_neurons[i]);
     }
  }

  cudaMalloc((void**)&d_lay, sizeof(float) * totallay);
```

Fig. 20: Al·locatació de memòria per al Device

A.4.3 cudaMemcpy()

Apèndix vinculat a les figures 21 i 22.

```
//d_input
      for(int i = 0; i<NUMPAT; i++)
      {
            cudaMemcpy(&d_input[i*NUMIN], input[i],
NUMIN*sizeof(char), cudaMemcpyHostToDevice);
      }

//d_desired_outputs
  cudaMemcpy(d_desired_outputs, desired_outputs,
NUMPAT*NUMOUT*sizeof(float), cudaMemcpyHostToDevice);

//d_lay
  int *displacement = (int*)malloc(num_layers*sizeof(int));
  int *access = (int*)malloc(num_layers*sizeof(int));
  int *access_weights = (int*)malloc(num_layers*sizeof(int));

  displacement[0] = 0;
  access[0] = num_neurons[0];
  access_weights[0] = num_neurons[0] * num_neurons[1];

  for(int i = 1; i < num_layers; i++)
  {
      displacement[i] = displacement[i-1] + 6 * num_neurons[i-1]
+ 2 * num_neurons[i] * num_neurons[i - 1];
      access[i] = num_neurons[i];
      if(i < num_layers - 1)
        access_weights[i] = num_neurons[i] * num_neurons[i+1];
  }

  cudaMalloc((void**)&d_displacement, num_layers*sizeof(int));
  cudaMalloc((void**)&d_access, num_layers*sizeof(int));
  cudaMalloc((void**)&d_access_weights, num_layers*sizeof(int));


  cudaMemcpy(d_displacement, displacement, num_layers*sizeof(int),
cudaMemcpyHostToDevice);
  cudaMemcpy(d_access, access, num_layers*sizeof(int),
cudaMemcpyHostToDevice);
  cudaMemcpy(d_access_weights, access_weights,
num_layers*sizeof(int), cudaMemcpyHostToDevice);


  for(int i = 0; i < num_layers; i++)
  {
      cudaMemcpy(&d_lay[displacement[i]], lay[i].actv,
num_neurons[i]*sizeof(float), cudaMemcpyHostToDevice);
      cudaMemcpy(&d_lay[displacement[i] + access[i]], lay[i].bias,
num_neurons[i]*sizeof(float), cudaMemcpyHostToDevice);
      cudaMemcpy(&d_lay[displacement[i] + access[i] * 2], lay[i].z,
num_neurons[i]*sizeof(float), cudaMemcpyHostToDevice);
      cudaMemcpy(&d_lay[displacement[i] + access[i] * 3],
lay[i].dactv, num_neurons[i]*sizeof(float), cudaMemcpyHostToDevice);
      cudaMemcpy(&d_lay[displacement[i] + access[i] * 4],
lay[i].dbias, num_neurons[i]*sizeof(float), cudaMemcpyHostToDevice);
      cudaMemcpy(&d_lay[displacement[i] + access[i] * 5],
lay[i].dz, num_neurons[i]*sizeof(float), cudaMemcpyHostToDevice);
```

Fig. 21: Copia dels atributs del Host al Device (primera part)

```
    if(i < num_layers - 1)
    {
      cudaMemcpy(&d_lay[displacement[i] + access[i] * 6],
lay[i].out_weights, num_neurons[i]*num_neurons[i+1]*sizeof(float),
cudaMemcpyHostToDevice);
      cudaMemcpy(&d_lay[displacement[i] + access[i] * 6 +
access_weights[i]], lay[i].dw,
num_neurons[i]*num_neurons[i+1]*sizeof(float),
cudaMemcpyHostToDevice);
    }
  }
```

Fig. 22: Copia dels atributs del Host al Device (segona part)

A.4.4 Kernel feed

Apèndix vinculat a la Figura 23.

```
//------Kernel feed------
__global__ void feed(int p, char *d_input, float *d_lay)
{
        //Feed inputs to input layer
        int i = threadIdx.x; //NUMIN
        d_lay[i] = d_input[p*NUMIN + i]*1.0;
}
```

Fig. 23: Kernel feed

A.4.5 Kernel forward

Apèndix vinculat a la Figura 24.

```
//----- Kernel forward_prop-----
extern __shared__ float d_SumZ[];
__global__ void forward(float *d_lay, int layer, int next, int
num_neurons_bef, int *displacement, int *access)
{
  int col = threadIdx.x;
  int fila = blockIdx.x;
  if(col == 0)
    d_lay[displacement[layer] + access[layer] * 2 + fila] =
d_lay[displacement[layer] + access[layer] + fila];

  d_SumZ[col] = d_lay[(displacement[layer-1] + access[layer-1] * 6)
+ fila * num_neurons_bef + col] * d_lay[displacement[layer-1] +
col];

  for(int stride = next/2; stride > 0; stride/=2)
  {
      __syncthreads();
     if((col < stride) && (col+stride < num_neurons_bef))
d_SumZ[col] += d_SumZ[col + stride];
  }

  if(col == 0)
  {
    d_lay[displacement[layer] + access[layer] * 2 + fila] +=
d_SumZ[0];
  }

  __syncthreads();

  if(col== 0)
    d_lay[displacement[layer] + fila] = (layer < 2)?
((d_lay[displacement[layer] + access[layer] * 2 + fila] < 0) ? 0 :
d_lay[displacement[layer] + access[layer] * 2 + fila]) : 1/(1+exp(-
d_lay[displacement[layer] + access[layer] * 2 + fila]));

}
```

Fig. 24: Kernel forward

A.4.6 Kernel back_first

Apèndix vinculat a la Figura 25.

```
//----- Kernels Back_prop-----
__global__ void back_first(float *d_lay, float *d_desired_outputs,
int p, int layer, int *displacement, int *access)
{
  int col = threadIdx.x;
  d_lay[displacement[layer] + access[layer] * 5 + col] =
(d_lay[displacement[layer] + col] - d_desired_outputs[p*NUMOUT +
col]) * d_lay[displacement[layer] + col] * (1 -
d_lay[displacement[layer] + col]);
  d_lay[displacement[layer] + access[layer] * 4 + col] =
d_lay[displacement[layer] + access[layer] * 5 + col];

}
```

Fig. 25: Kernel back_first

A.4.7 Kernel back_second

Apèndix vinculat a la Figura 26.

```
__global__ void back_second(float *d_lay, float *d_desired_outputs,
int p, int num_neurons_bef, int num_layers, int *displacement, int
*access, int *access_weights)
{
  int col = threadIdx.x;
  int fila = blockIdx.x;

  d_lay[(displacement[num_layers-2] + access[num_layers-2] * 6 +
access_weights[num_layers-2]) + fila * num_neurons_bef + col] =
d_lay[displacement[num_layers-1] + access[num_layers-1] * 5 + fila]
* d_lay[displacement[num_layers-2] + col];
  __syncthreads();

  if(fila == 9)
    d_lay[displacement[num_layers-2]+ access[num_layers-2] * 3 +
col] =  d_lay[(displacement[num_layers-2] +  access[num_layers-2] *
6) + fila * num_neurons_bef + col] * d_lay[displacement[num_layers-
1] +  access[num_layers-1] * 5 + fila];
}
```

Fig. 26: Kernel back_second

A.4.8 Kernel back_third

Apèndix vinculat a la Figura 27.

```
__global__ void back_third(float *d_lay, int num_neurons_bef, int
layer, int *displacement, int *access, int *access_weights)
{

  int col = threadIdx.x;
  int fila = blockIdx.x;

  d_lay[displacement[layer] + access[layer] * 5 + fila] =
(d_lay[displacement[layer] + access[layer] * 2 + fila] >= 0)?
d_lay[displacement[layer] + access[layer] * 3 + fila]: 0;
  d_lay[(displacement[layer-1] + access[layer-1] * 6 +
access_weights[layer-1]) + fila * num_neurons_bef + col] =
d_lay[displacement[layer] + access[layer] * 5 + fila] *
d_lay[displacement[layer-1] + col];
  d_lay[displacement[layer-1] + access[layer-1] * 3 + col] = (layer
> 1)? (d_lay[(displacement[layer-1] + access[layer-1] * 6) + fila *
num_neurons_bef + col] * d_lay[displacement[layer]+ access[layer] *
5 + fila]) : d_lay[displacement[layer-1] + access[layer-1] * 3 +
col] ;
  d_lay[displacement[layer] + access[layer] * 4 + fila] =
d_lay[displacement[layer]+ access[layer] * 5 + fila];

}
```

Fig. 27: Kernel back_third

A.4.9 Kernel update

Apèndix vinculat a la Figura 28.

```
//----- Kernel Update_weights-----
__global__ void update(float *d_lay, int num_neurons_act, int
num_neurons_next, float alpha, int layer, int *displacement, int
*access, int *access_weights)
{

  int col = threadIdx.x;
  int fila = blockIdx.x;

  d_lay[(displacement[layer] + access[layer] * 6) + fila *
num_neurons_act + col] = d_lay[(displacement[layer] + access[layer]
* 6) + fila * num_neurons_act + col] - (alpha *
d_lay[(displacement[layer] + access[layer] * 6 +
access_weights[layer]) + fila * num_neurons_act + col]);

  if(fila == 0)
  {
    d_lay[displacement[layer] + access[layer] + col] =
d_lay[displacement[layer] + access[layer] + col] - (alpha *
d_lay[displacement[layer] + access[layer] * 4 + col]);
  }
}
```

Fig. 28: Kernel update