UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desenvolupament de processos d'entrenament i inferència amb aritmètica en coma flotant de 16 bits sobre la plataforma Jetson Xavier

TREBALL FI DE GRAU

Grau en Enginyeria Informàtica

Autor: Díaz - Cano Lozano, Roberto

Tutors: Flich Cardo, José,
Molero Prieto, Xavier

Curs 2020-2021

Resum

La intel·ligència artificial (IA) s'està convertint en un element imprescindible en diferents àmbits de la informàtica. Al mateix temps que la IA s'està desenvolupant a escala d'algoritmes, també les arquitectures de processament s'hi estan adaptant per donar un millor suport. Per aquest motiu en el present treball es desenvolupa un suport en aritmètica en coma flotant de 16 bits sobre una plataforma d'entrenament i inferència de xarxes neuronals. Aquest desenvolupament es realitza sobre el dispositiu Jetson AGX Xavier de l'empresa NVIDIA, el qual està destinat a aplicacions d'intel·ligència artificial, com ara l'aprenentatge profund (*deep learning*).

L'objectiu és dotar a l'aplicació HELENNA d'un suport que li permeta utilitzar els nombres en coma flotant de 16 bits sobre la GPU del dispositiu de NVIDIA, a través del llenguatge de programació CUDA. D'aquesta manera es podrà aconseguir un millor aprofitament dels recursos i del consum energètic, ja que amb l'aritmètica de precisió reduïda es pot incrementar l'eficiència dels entrenaments amb xarxes neuronals.

Paraules clau: Intel·ligència artificial, xarxes neuronals, HELENNA, GPU, aritmètica en coma flotant de 16 bits, CUDA, aprenentatge profund.

Resumen

La inteligencia artificial (IA) se está convirtiendo en un elemento imprescindible en diferentes ámbitos de la informática. Al mismo tiempo que la IA se está desarrollando a escala de algoritmos, también las arquitecturas de procesamiento se están adaptando para dar un mejor apoyo. Por este motivo en el presente trabajo se desarrolla un soporte en aritmética en coma flotante de 16 bits sobre una plataforma de entrenamiento e inferencia de redes neuronales. Este desarrollo se realiza sobre el dispositivo Jetson AGX Xavier de la empresa NVIDIA, el cual está destinado a aplicaciones de inteligencia artificial, como el aprendizaje profundo (*deep learning*).

El objetivo es dar a la aplicación HELENNA de un soporte que le permita utilizar nombres en coma flotante de 16 bits sobre la GPU del dispositivo de NVIDIA, a través del lenguaje de programación CUDA. De este modo se podrá conseguir un mejor aprovechamiento de los recursos y del consumo energético, puesto que con la aritmética de precisión reducida se puede incrementar la eficiencia de los entrenamientos con redes neuronales.

Palabras clave: Inteligencia artificial, redes neuronales, HELENNA, GPU, aritmética en coma flotante de 16 bits, CUDA, aprendizaje profundo.

Abstract

Artificial Intelligence (AI) is becoming an important element in different areas of computing. At the same time that AI is developing at the algorithm scale, processing architectures are also adapting to give a better support to it. For this reason, in the present project a 16-bit floating point arithmetic support is developed on a neural network training and inference platform. This development is carried out on the Jetson AGX Xavier device from NVIDIA, which is destined for artificial intelligence applications, like deep learning.

The objective is to give the HELENNA application support that allows it to use 16-bit floating point numbers on the GPU of the NVIDIA device, through the CUDA programming language. In this way, better use of resources and energy consumption can be achieved, since with reduced precision arithmetic, the efficiency of training with neural networks can be increased.

Key words: Artificial Intelligence, neuronal networks, HELENNA, GPU, 16-bit floating point arithmetic, CUDA, deep learning.

Índex

Índex de figures

Índex de taules

CAPÍTOL 1
Introducció

Aquest treball final de grau es centra en una de les tecnologies que més s'han desenvolupat al llarg dels anys, el *deep learning*. Concretament en l'entrenament de xarxes neuronals i la seua inferència. En aquest capítol es desenrotllen els objectius que es pretenen complir una vegada finalitzat el projecte. A més, es descriu el context de treball, els Objectius de Desenvolupament Sostenible que compleix el treball i l'estructura que posseeix la memòria. Per últim, s'explicarà com s'han utilitzat totes les fonts bibliogràfiques.

1.1 Motivació

La intel·ligència artificial (IA) té l'objectiu de fer que les màquines siguen capaces d'actuar d'acord amb els estímuls obtinguts a través de diferents dispositius, com poden ser imatges, arxius de so, etc. Per tant, s'encunya aquest terme per tal de designar a qualsevol màquina que proporcione la sensació de raonar, aprendre, percebre o resoldre problemes per ella sola. La IA no és una tecnologia que es va inventar fa uns pocs anys. Aquest terme es remunta al segle XIV, quan es desenvolupà la lògica matemàtica, però on realment va obtenir un important ressò va ser quan en 1950 Alan Turing presentà un article on proposà una prova per a determinar si una màquina era intel·ligent o no, la Prova de Turing. Aquesta prova consistia en el fet que, si una màquina era capaç de contestar a unes preguntes igual que un humà, aleshores es podia considerar que era "intel·ligent". Amb aquest article Alan Turing es convertí en el referent d'aquest camp i el seu estudi va ser un dels més importants a l'hora de desenvolupar la IA.

Hui en dia, la intel·ligència artificial s'ha desenvolupat en diferents rames, com els sistemes que simulen el pensament humà i per tant dona la sensació que pensen com nosaltres, un d'aquests sistemes serien les xarxes neuronals artificials; o els que actuen com a humans, com els cotxes autònoms (veure Figura 1.1).

El present treball es concentra en una d'aquestes ramificacions de la IA, les xarxes neuronals. Estan inspirades en les neurones dels sistemes nerviosos dels humans. Aquestes xarxes consisteixen en un grup de neurones, connectades totes entre si, que treballen en conjunt. Cada neurona és una unitat de càlcul que, donada una entrada, produeix una eixida que és enviada a la següent neurona (veure Figura 1.2). Aquest sistema entra dins del paradigma del *deep learning*, que consisteix a aconseguir que un computador siga capaç d'aprendre sense cap supervisió.

La utilització d'aplicacions que les fan servir són cada vegada més comuns al món tecnològic. Açò es deu al fet que són molt útils en tasques de reconeixement d'imatges o visió per computador. Un altre factor que ha ajudat en la popularització de les xarxes neuronals ha sigut la Unitat de Processament Gràfic (*Graphics Processing Unit*, GPU), ja

Figura 1.1: Vehicle que a través de les dades que obté dels sensors és capaç de conduir i prendre decisions per ell sol en qüestió de segons.

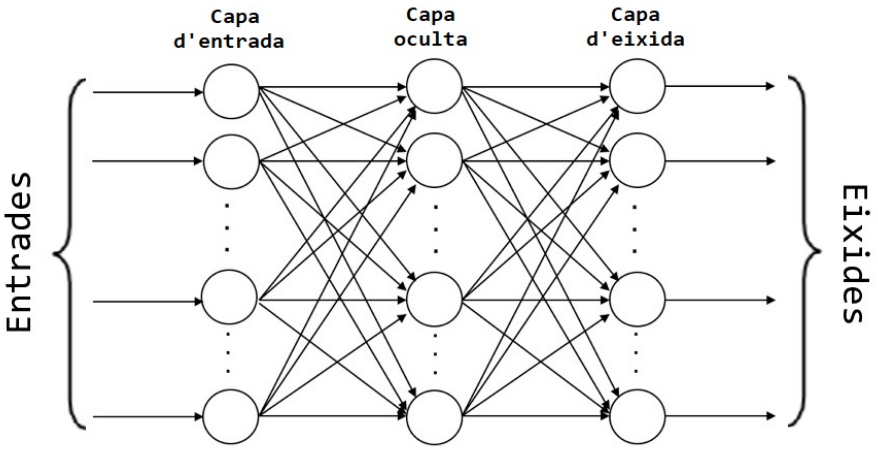

Figura 1.2: Exemple de xarxa neuronal amb una capa d'entrada, una oculta i una d'eixida. La capa oculta es pot compondre per moltes més capes de neurones, depenent de la implementació de la xarxa.

que aquest dispositiu permet realitzar entrenaments més ràpidament que amb la Unitat de Processament Central (*Central Processing Unit*, CPU). Aquest fet és causat per la gran quantitat de nuclis que tenen les GPU, fet que els atorga una gran capacitat de còmput respecte a les CPU (veure Figura 1.3). Una de les altres raons és per la implementació de biblioteques i programari com ara cuBLAS o cuDNN, que permeten obtenir un major rendiment d'aquests dispositius.

Dins d'aquestes aplicacions es troba HELENNA, desenvolupada per a l'entrenament i inferència de xarxes neuronals, destinada principalment per a treballs relacionats amb la docència i la investigació. HELENNA és una aplicació amb una computació heterogènia, ja que té suport per a ser executada en CPU i GPU, entre altres, i utilitza diferents biblioteques com cuBLAS, MKL o OpenCV. Açò fa que l'aplicació es puga utilitzar en la gran part de sistemes que hi ha hui en dia al mercat.

La implementació del suport per a GPU en HELENNA es va dissenyar per donar suport tant a les gràfiques d'AMD com a les de NVIDIA. Pel que respecta a les targetes gràfiques del fabricant NVIDIA, el programari dissenyat per a l'aplicació se centra a explotar una de les últimes microarquitectures que presenten les seues GPU, l'anomenada microarquitectura Volta. Amb aquesta microarquitectura el fabricant va posar el focus

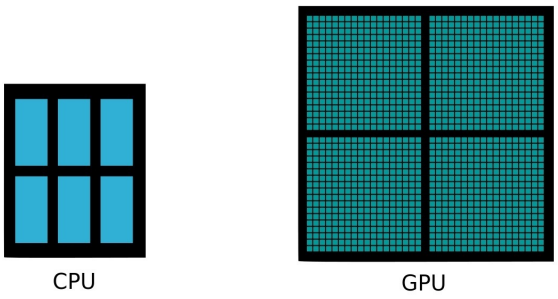

Figura 1.3: Diferencia gràfica dels nuclis que formen la CPU i la GPU. Com s'observa a la imatge la CPU sol comptar amb molt pocs nuclis respecte a la GPU que pot albergar milers de nuclis. Aquest fet fa que la GPU tinga una alta capacitat de còmput, fent-la idònia per a les tasques d'intel·ligència artificial.

en la intel·ligència artificial, dotant les targetes gràfiques de major capacitat d'operacions amb nombres reals de simple precisió, amb el qual augmenta considerablement el rendiment del *deep learning* i per conseqüència, de les xarxes neuronals. Per altra banda, també té suport a les biblioteques, com ara cuBLAS[1] o cuDNN[2], i al llenguatge de programació CUDA[3]. Amb açò el fabricant aconsegueix que les GPU siguen molt atractives per a qualsevol aplicació enfocada en la IA. Una de les altres raons per la que es va escollir aquesta microarquitectura és perquè amb aquesta versió també es poden utilitzar els nombres reals de precisió reduïda, és a dir, són nombres reals, però tenen la meitat de precisió que els nombres reals de simple precisió. Per tant, aquest fet fa que els nombres siguen més menuts i el temps de còmput siga menor. En canvi, la utilització d'aquest tipus de dada comporta una pèrdua de precisió en els entrenaments de les xarxes neuronals.

Per a donar suport a les targetes gràfiques amb la microarquitectura mencionada anteriorment a HELENNA s'ha utilitzat la NVIDIA Jetson AGX Xavier (veure Figura 1.4), ja que aquest dispositiu, a banda de tenir una mida menuda, té una gran capacitat per al còmput. Aquest dispositiu està compost per una GPU amb microarquitectura Volta i 512 nuclis, dos acceleradors de *deep learning* i una alta capacitat de transferència de dades des de GPU a CPU o viceversa, fent-la idònia per a l'entrenament i inferència. Per altra banda, aquest dispositiu també permet a l'usuari configurar el consum energètic, permetent que per a determinades situacions on no siga necessària una elevada potència de càlcul es puga utilitzar un mode energètic que oferisca menys potència, però també que consumisca menys, fent que el dispositiu es puga adaptar a cada tipus de treball de forma òptima.

1.2 Objectius

L'objectiu general del present treball consisteix a donar suport a l'aplicació HELENNA perquè faça ús dels nombres reals de mitja precisió, mencionats a la secció anterior, a través del dispositiu NVIDIA Jetson AGX Xavier. Per assolir-ho s'ha intentat explotar el millor possible tots els recursos disponibles, és a dir, la computació paral·lela que ofereixen les GPU i l'ús de la biblioteca de càlcul d'àlgebra lineal, cuBLAS.

[1] https://developer.nvidia.com/cublas
[2] https://developer.nvidia.com/cudnn
[3] https://developer.nvidia.com/cuda-zone

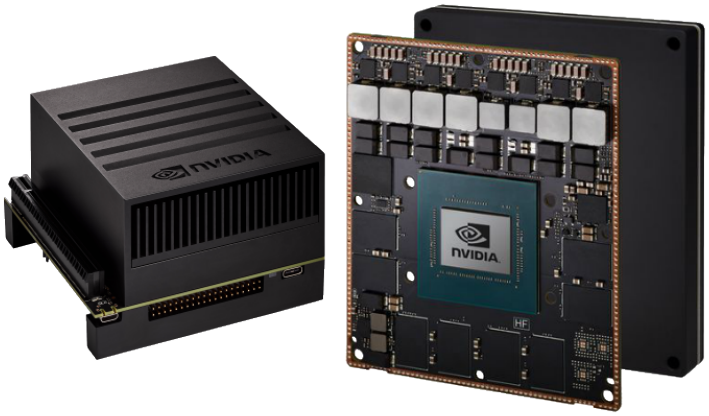

Figura 1.4: Dispositiu NVIDIA Jetson AGX Xavier. Amb aquesta imatge es pretén il·lustrar com és el dispositiu per fora i com és la placa base amb tots els components que en conjunt fan funcionar el dispositiu.

Per aconseguir-ho, s'han implementat les funcions necessàries per al correcte funcionament de les diferents capes que componen l'aplicació. Aquestes capes són:

- La **capa completament connectada**, on totes les neurones estan connectades entre si. L'objectiu d'aquesta capa és predir l'objecte que hi ha representat en una imatge, entre d'altres.
- La **capa de convolució**, amb l'objectiu principal del reconeixement d'imatges.
- La **capa de normalització**, les anomenades *dropout* i *batch normalization*. Aquestes capes s'encarreguen de normalitzar les dades intermèdies del procés d'execució per aconseguir un entrenament de major qualitat.

Altrament, també s'ha fet un estudi del consum energètic del dispositiu NVIDIA Jetson AGX Xavier per a poder veure l'impacte energètic que té HELENNA i com el podem disminuir. Aquesta anàlisi s'ha fet executant l'aplicació amb la mateixa base de dades, però canviant el mode de potència del dispositiu. Aquestes execucions s'han repetit amb totes les bases de dades disponibles per a poder observar quin mode d'energia s'adapta millor a cada base de dades.

Per últim, s'ha realitzat un estudi a fi de veure la millora que s'obté per utilitzar els nombres reals de mitja precisió en l'aplicació. Per tal d'obtenir-ho s'ha comparat els temps d'execució i el consum energètic de l'aplicació quan s'executa amb nombres reals de precisió simple i amb nombres reals de precisió reduïda.

1.3 Context del treball

El treball fi de grau està basat en el treball realitzat en les pràctiques d'empresa en el Grup d'Arquitectures Paral·leles (GAP), integrat en el Departament d'Informàtica de Sistemes i Computadors (DISCA) de la Universitat Politècnica de València.

Les pràctiques s'han efectuat en un grup de treball format per diferents membres del grup d'investigació del GAP i per altres alumnes que també estaven realitzant les pràctiques d'empresa. Al tenir tots un treball relacionat amb l'aplicació HELENNA, s'ha pogut treballar en equip i també compartir informació per ajudar-nos mútuament a resoldre els diferents conflictes trobats durant la realització del treball. Com que les pràctiques

es realitzaven a distància, hem tingut a la nostra disposició l'aplicació Slack per poder comunicar-nos entre nosaltres i fer qüestions ràpides als tutors. Tanmateix, hem celebrat reunions setmanals a través de la plataforma Microsoft Teams per exposar les tasques complides, les dades obtingudes i resoldre els dubtes ocasionats al llarg de la setmana. Un altre dels objectius d'aquestes reunions ha sigut fer un control del treball realitzat pels alumnes, on els tutors ho aconseguien marcant diferents objectius que havíem de complir per a la següent reunió.

Com a conseqüència, la metodologia executada pels tutors m'ha ajudat a millorar en diferents aspectes, com ara treballar en equip o organitzar-me millor, entre altres.

Per un altre costat, també cal destacar que per a la realització d'aquest treball han sigut molt útils diferents assignatures estudiades en el Grau en Enginyeria Informàtica com ara:

- **Àlgebra (ALG)**, per ser una matèria on s'aborden les operacions matricials.
- **Estructura de Computadors (ETC)**, per oferir una visió generalitzada de com funcionen els processadors i el sistema de memòria.
- **Arquitectures Avançades (AAV)**, per oferir una versió més especifica del funcionament dels processadors i per ensenyar a fer anàlisis de dades, obtingudes a partir de l'execució d'un programa.
- **Computació paral·lela (CPA)** i **Llenguatges i entorns de programació paral·lela (LPP)**, aquestes assignatures estan molt relacionades entre si i en les dues s'aborda la programació paral·lela amb el llenguatge C/C++.
- **Disseny de Sistemes Operatius (DSO)**, per oferir una visió de com implementar fitxers de compilació com ara `make` i `cmake`, ambdós útils per a facilitar la incorporació d'un nou mòdul a l'aplicació HELENNA.
- **Sistemes Intel·ligents (SIN)**, per ser una matèria on s'aborda la intel·ligència artificial i explicar el funcionament del perceptró, les xarxes neuronals, etc.

1.4 Objectius de Desenvolupament Sostenible (ODS)

Els ODS estàn compostos per 17 objectius (veure Figura 1.5), els quals van ser dissenyats per l'Assemblea General de les Nacions Unides amb la finalitat de marcar les estratègies de desenvolupament a escala mundial durant els pròxims quinze anys. Amb aquests objectius es pretén erradicar la pobresa, protegir el planeta i assegurar la prosperitat, entre d'altres.

Amb el present treball fi de grau es vol complir amb dos dels objectius marcats per les Nacions Unides:

- **ODS 7. Ús eficient de l'energia**. Aquest objectiu es compleix mitjançant l'ús dels diferents modes d'energia de NVIDIA Jetson AGX Xavier. Amb aquest dispositiu es pot seleccionar el mode que més s'adapte a la tasca a realitzar per estalviar energia i fer l'aplicació més sostenible.
- **ODS 9. Fomentar la investigació**. HELENNA pot incentivar la investigació, ja que pot ajudar els investigadors a dur a terme els seus projectes i obri la possibilitat de desenvolupar noves tasques de diferents maneres. A més, com que és una aplicació de computació heterogènia, pot ser utilitzada en una gran varietat de dispositius, tot fent-la idònia per a qualsevol entorn de treball.

Figura 1.5: Els 17 objectius que formen els ODS dissenyats per l'ONU amb la finalitat de millorar la qualitat de vida de tots els ciutadans. On l'organització posa l'atenció en una millora en la salut, l'educació i el benestar dels països subdesenrotllats, com també fomentar la investigació i reduir el canvi climàtic.

1.5 Estructura de la memòria

A continuació es detalla la composició de la memòria i es descriu breument el contingut dels cinc capítols que la componen. Aquests capítols són:

1. **Introducció**: en aquest capítol s'explica la motivació per fer el present treball, els objectius a aconseguir i el context de treball on s'ha desenvolupat. A més, s'exposen els objectius de desenvolupament sostenible que es compleixen i l'ús de la bibliografia.

2. **Entorn de treball**: en aquesta secció s'indiquen totes les eines que han sigut utilitzades al llarg del projecte. Es detallen les característiques del dispositiu NVIDIA Jetson AGX Xavier. S'explica la Unitat de Processament Gràfic i en què consisteix, també es detallen alguns aspectes de l'empresa NVIDIA. Després, és donen detalls sobre la plataforma CUDA i les biblioteques utilitzades, l'aplicació HELENNA i els nombres reals de mitja precisió. A més, es mostra d'una forma més detallada les xarxes neuronals, així com les aplicacions que s'han utilitzat per identificar els possibles errors d'implementació.

3. **Contribució a HELENNA**: aquest capítol se centra a presentar i explicar les diferents capes que s'han implementat en HELENNA. D'altra banda, també es comentarà la metodologia emprada per a fer la implementació de les funcions, així com tots els inconvenients trobats al llarg del treball i com s'han solucionat. Per últim, en la recta final del treball es feren diferents optimitzacions d'algunes funcions, les quals s'exposen en l'última secció del capítol. En aquesta secció es mostra el treball que s'ha fet i les dades obtingudes, tant abans com després de realitzar-les.

4. **Resultats obtinguts en HELENNA**: en el quart capítol es presenten els resultats obtinguts amb HELENNA en totes les capes implementades. Aquest capítol està dividit en tres parts: una on es veuen xarxes neuronals que utilitzen només una

capa oculta, una altra on es veu una xarxa neuronal que utilitza totes les capes explicades en el capítol 3 i un estudi per veure l'impacte que té la mida dels blocs en l'execució de l'aplicació. Amb aquestes dades es fa una anàlisi exhaustiva per veure l'eficiència temporal i de consum energètic del nou mòdul implementat tant.

5. **Conclusions i treball futur**: en aquest apartat es presenten les conclusions obtingudes amb la realització del treball i es plantegen diverses formes d'ampliar-lo en el futur.

1.6 Notes bibliogràfiques

En aquesta secció es detalla com s'ha utilitzat la bibliografia per poder exposar tots els conceptes que s'expliquen al llarg de la memòria.

En primer lloc, el capítol 2 és el que més referències bibliogràfiques té de tots els capítols que formen aquesta memòria, ja que aquest se centra a explicar tots els conceptes necessaris per al correcte enteniment del treball. La primera part del capítol, el qual explica la GPU i la seua història, s'ha utilitzat principalment el llibre *Programming Massively Parallel Processors* ([13]) i s'ha complementat la informació amb les pàgines web següents: [32, 5, 4, 11, 12]. Després, per a comparar el mode de funcionament de les CPU i les GPU s'ha utilitzat el llibre [8] i les pàgines web [34, 3]. Per a detallar la trajectòria de l'empresa NVIDIA s'ha utilitzat [9] i per explicar el dispositiu AGX Xavier i la diferència amb les seues antecessores s'ha emprat les referencies [24, 27].

En la segona part del capítol, en la secció de les xarxes neuronals on s'exposa com funcionen, s'ha utilitzat el llibre [7, 16]. A més, per exposar els nombres reals de simple i mitja precisió s'han fet servir el llibres llibres [10, 35, 26] i la pàgina web [25]. A continuació, per explicar CUDA, el seu model de programació i les biblioteques que s'han utilitzat, s'han consultat els llibres [30, 2, 1, 33] i les pàgines web [17, 18], d'aquests cal destacar el *CUDA C Programming*, per ser un dels més complets a l'hora d'explicar la plataforma CUDA. En aquest cas s'han seleccionat les seccions dels llibres i pàgines web que millor explicaven allò que es volia exposar en la memòria. Per últim, per explicar les eines per comprovar possibles errors d'implementació i compartir el treball amb els membres del grup, s'han utilitzat [22, 19, 23].

En segon lloc, el capítol 3 s'exposa el treball realitzat en HELENNA i s'ha fet referència a [6, 28, 21, 20, 17]. En aquest cas s'han utilitzat per explicar les diferents operacions aritmètiques i funcions de la biblioteca cuBLAS. A més, amb [15, 7, 31, 29] s'han definit la capa altament connectades, la capa convolucional i la de normalització i diferents conceptes que no s'han detallat en el capítol 2.

Per últim, en el capítol 4 s'ha utilitzat [14] per indicar d'on s'ha obtingut la configuració dels diferents modes de potència de l'AGX Xavier.

CAPÍTOL 2
Entorn de treball

En aquest capítol es presenta l'entorn de treball i totes les eines que s'han utilitzat per desenvolupar-lo. S'explicarà en què consisteix la GPU, que és NVIDIA i les característiques del seu dispositiu. A més, es detallaran les xarxes neuronals, què és l'aplicació HELENNA i l'entorn de programació CUDA, el tipus de dada i les biblioteques que s'han utilitzat. Per últim, s'exposaran les eines emprades per identificar els possibles errors d'implementació i per canviar els modes de potència de NVIDIA Jetson AGX Xavier.

2.1 NVIDIA Jetson AGX Xavier

En aquest apartat s'explica què són les GPU, així com la seua història. També s'explica en què consisteix l'empresa NVIDIA i, el més important de tot, les característiques de NVIDIA Jetson AGX Xavier i per què és un dels millors dispositius per a l'execució d'aplicacions d'inferència.

2.1.1. Unitat de Processament Gràfic

La Unitat de Processament Gràfic és un processador que suplementa les operacions de la CPU, dedicat al processament d'imatges i operacions de coma flotant. Aquest té la finalitat d'ajudar al processador central en determinades aplicacions que tenen un alt nombre d'operacions amb nombres reals. Els processadors s'incorporen dins d'unes targetes, les quals anomenem targetes gràfiques. Aquestes són una extensió de la placa base de l'ordinador amb diferents components que permeten el correcte funcionament de la GPU. (veure Figura 2.1).

La GPU ha sigut un dels components que més ha evolucionat al llarg dels anys del segle XX. Per entendre com són les GPU hui en dia ens hem de remuntar als anys setanta, quan l'empresa Arcade System Games utilitzà un component el qual representava la primera idea d'unitat de processament gràfic. Aquest consistia en uns circuits gràfics per a ajudar a la CPU a representar les imatges en una pantalla, ja que aquests circuits, per si mateixos, no eren capaços de representar figures en la pantalla. Gràcies a aquesta nova tecnologia es va poder fer una gran quantitat de jocs arcade com ara *Space Invaders* (veure Figura 2.2). Al cap de pocs anys, diverses empreses del sector dels videojocs, com ara Atari [4], presentaren els primers xips que podien enviar la imatge i el so als televisors. Amb aquesta tecnologia es pogueren llançar les primeres consoles de videojocs domèstiques.

Després, en la dècada dels vuitanta es crearen els primers xips, semblants a la GPU de hui en dia, de la mà de Number Nine Visual Technology, gràcies a NEC 7220 (veure Figu-

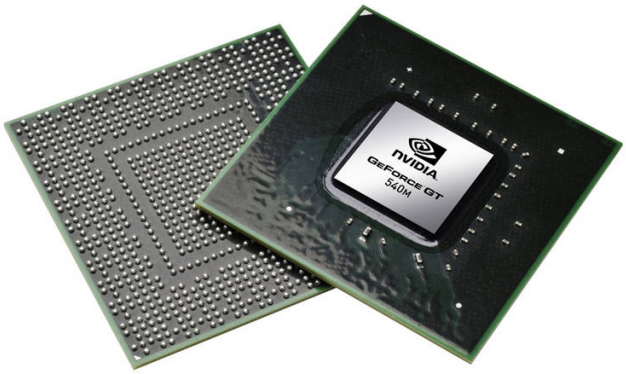

Figura 2.1: Unitat de processament gràfic de l'empresa NVIDIA. En aquest cas es correspon al model GeForce GT 540M.

Figura 2.2: El joc *Space Invaders* és un dels més importants en la història dels videojocs, es va llançar el 1978 en les màquines recreatives i va ser tot un èxit, fins al punt que hi havia competicions per veure qui era capaç d'obtenir la puntuació més elevada.

ra 2.3), un processador capaç de dibuixar en pantalla. A partir d'aquest moment, altres empreses com ara Intel [32] començaren a desenvolupar i presentar les unitats gràfiques per als ordinadors. Però, la que més va destacar durant aquest període va ser l'empresa IBM per ser capaç de crear imatges amb la resolució més elevada de l'època.

A principis dels noranta la majoria dels ordinadors ja disposaven de xips gràfics amb acceleració 2D, la qual cosa donà peu a que els sistemes operatius integraren la interfície gràfica de l'usuari (*graphical user interface*, GUI)[1]. En la meitat d'aquesta dècada es va començar a implementar la computació gràfica 3D en temps real, tot permetent obtenir un gran avanç en els ordinadors i les consoles. Però aquesta tecnologia no estava integrada amb l'acceleració 2D, per tant feia que la utilització dels models 3D fora molt limitada. Com a conseqüència, sols foren utilitzades en videoconsoles com la Nintendo 64 (Figura 2.4). Després, es presentaren els primers xips de l'empresa Redention amb la possibilitat de tenir en un mateix dispositiu l'acceleració 2D i 3D. A més, es desenvoluparen distintes interfícies de programació d'aplicacions (*application programming interface*, API)[2]

[1] Interfície que permet als usuaris interactuar amb els dispositius electrònics a través d'icones gràfiques i indicadors de so.

[2] Interfície composta per una col·lecció de funcions de biblioteca que permeten a les aplicacions utilitzar serveis i funcionalitats de programari i maquinari.

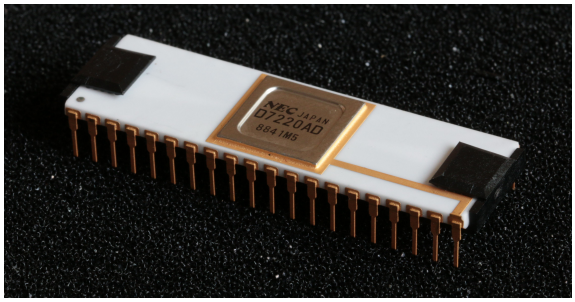

Figura 2.3: El component NEC 7220 fou el primer xip controlador de gràfics integrats que era capaç de dibuixar caràcters, línies, cercles, etc. en una pantalla utilitzant un mapa de bits com a referència [5].

([13], pàgines 23-39) com ara DirectX, creada per Microsoft, centrada en tasques multimèdia, on tan sols es podia gastar amb aplicacions de l'ecosistema de la companyia. D'altra banda, també es creà OpenGL, una interfície multiplataforma i de codi obert per ajudar els desenvolupadors a crear aplicacions en altres plataformes que no foren la de Microsoft. D'aquesta forma es donà oxigen a altres sistemes operatius, com Linux que també és de codi obert. En 1994, fou quan per primera vegada s'encunyà el terme de «GPU» per referir-se a la unitat de processador de geometria o *Geometry Processor Unit* per l'empresa Sony quan va fer referència al sistema de creació dels models 3D en la presentació de la Play Station 1 [12]. Però en 1999, quan NVIDIA presentà la GeForce 256, és quan verdaderament s'encunyà el terme d'Unitat de processament Gràfic tal com el coneixem hui en dia.

Figura 2.4: La Nintendo 64 va ser una de les primeres consoles domèstiques del mercat en tenir videojocs en 3 dimensions com Super Mario 64. Es deien així per utilitzar 64 bits de potència gràfica. Per a més informació veure [11].

En la dècada dels 2000 va ser l'època on l'empresa NVIDIA presentà el primer xip capaç de realitzar un ombreig de píxels programables. Aquesta tecnologia consisteix a programar l'ombreig de les imatges píxel a píxel. Aquest fet va permetre aconseguir unes ombres més complexes i d'una forma més ràpida, ja que en tractar píxels independents es podia fer de manera paral·lela. D'altra banda, uns anys després, es llançaren les noves versions d'OpenGL i DirectX que van permetre ampliar la capacitat de programació general de les GPU. Aquestes passaren a ser tan flexibles com les CPU, obtenint una computació més generalitzada. Per aconseguir-ho els investigadors s'adonaren que havien de convertir els problemes a resoldre en operacions gràfiques perquè el càlcul poguera iniciar-se a través de crides d'API d'OpenGL o DirectX. Per exemple, per a executar diverses funcions de càlcul paral·lelament, s'havien d'escriure com un ombreig de píxels.

Així mateix, les empreses del sector, en adonar-se d'aquest possible ús de les GPU, decidiren llançar, al final de la dècada, CUDA i OpenCL per a la programació en sistemes heterogenis. Amb aquestes biblioteques i compiladors fou possible expandir l'ús de les GPU a molts altres camps d'una manera més senzilla.

Des del 2010 fins l'actualitat, les targetes gràfiques han obtingut molt més ressò en tots els àmbits de la informàtica, ja que gràcies a les innovacions de la dècada anterior va ser possible donar-li uns altres usos a la GPU. Aquestes ja no tenien com a principal funció crear models 2D/3D. Igualment, gràcies a la seua estructura altament paral·lela li permet fer una gran quantitat d'operacions alhora. Açò era de gran ajuda per als desenvolupadors, ja que podien crear escenes realistes amb tècniques avançades d'il·luminació i ombrejat. A més, les GPU poden arribar a ser més eficients que les CPU en determinades situacions, com ara aquelles on cal processar una gran quantitat de blocs de dades. Amb aquests avanços tecnològics i les innovacions de diverses empreses del sector fou possible aconseguir GPU de dimensions més reduïdes, amb una gran capacitat de còmput i un baix consum energètic. Aquesta nova GPU ha permés accelerar dràsticament la càrrega de treball en computació d'alt rendiment, utilitzar-les per al reconeixement d'imatges i de la parla o l'entrenament de models d'inferència, instal·lar-les en vehicles autònoms, etc.

Com s'ha vist al llarg de la secció, la GPU es va crear amb l'objectiu d'ajudar a la CPU, però a poc a poc ha anat obtenint major rellevància en la informàtica fins a convertir-se en un element quasi igual d'important que la CPU. Sobretot, en aplicacions com el processament de dades, els videojocs, la visió de computador i el *deep learning*. En conseqüència, ara que les dues tenen una gran rellevància cal fer un incís per a veure quines són les diferències entre aquests dos components. Les principals són:

- La CPU ha de ser capaç d'executar i processar una gran varietat de treball, sobretot tasques en les quals la latència[3] o el rendiment del nucli tenen una gran importància. Altrament, la GPU se centra en activitats específiques de manera que no necessita tenir uns nuclis molt complexos [3].
- Els nuclis d'unitat aritmètica lògica (*arithmetic logic unit*, ALU)[4] que formen part de la CPU solen ser pocs, però molt complexos i amb molta memòria cau[5]. D'altra banda, la GPU es compon per milers de nuclis, però més senzills que els de la CPU [34] (veure Figura 2.5).
- La CPU executa les tasques de manera seqüencial, però el més ràpid possible, mentres que la GPU se centra més en la paral·lelització de les tasques, ja que en tenir milers de nuclis, les tasques es divideixen entre aquests obtenint una execució més ràpida.
- Les GPU permeten «relaxar» un dels problemes actuals al món de la informàtica com ara la llei de Moore. Aquesta llei diu que el nombre de transistors que es poden posar en un circuit integrat es duplicaran cada dos anys, però s'ha trobat una barrera física que fa que siga impossible seguir duplicant el nombre de transistors. Per tant, arribarà un moment en què serà impossible fer les CPU més ràpides, però amb el disseny de les GPU es pot seguir augmentant la capacitat de còmput sense augmentar el nombre de transistors [8].

[3] Temps que tarda el processador a executar una instrucció. Aquest temps pot variar depenent d'on es trobe ubicada la dada, ja que si la dada no està en la memòria cau, pot tardar molt més a obtenir-la i executar-la.

[4] L'ALU és un circuit digital que fa les operacions aritmètiques i lògiques amb nombres enters.

[5] Memòria de petita capacitat que guarda les dades més freqüents perquè en les futures sol·licituds d'aquestes dades s'obtinguen amb més rapidesa.

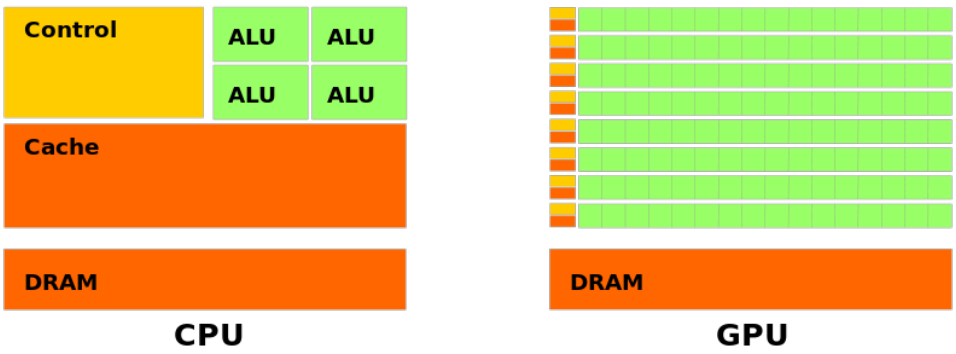

Figura 2.5: Diferencia gràfica dels components que formen la CPU i la GPU. Com s'observa a la imatge la CPU sol comptar amb molt pocs nuclis ALU respecte a la GPU, la qual pot albergar milers de nuclis. Per contra, la CPU compta amb una memòria cau molt més gran que la GPU.

2.1.2. NVIDIA

NVIDIA és una empresa tecnològica estatunidenca[6] especialitzada en el desenvolupament d'unitats de processament gràfic i circuits integrats. Al llarg dels anys, aquesta empresa s'ha buscat un lloc al mercat per ser una pionera en gràfiques i dispositius centrats en els videojocs, l'alta computació o la intel·ligència artificial. Un d'aquests dispositius és l'utilitzat en aquest treball, la NVIDIA Jetson AGX Xavier.

Aquesta empresa va ser fundada en 1993 amb l'objectiu que en algun moment els ordinadors domèstics podrien utilitzar-se per a executar jocs, música i imatges, cosa que en aquella època era quasi impossible [9]. La primera targeta gràfica que llançaren fou en 1995 i era una targeta amb connexió PCI[7] dotada d'un nucli per a gràfics en 2D/3D. Després, en 1996 va ser de les primeres companyies a donar suport a Microsoft DirectX. Amb açò oferiren la possibilitat de renderitzar[8] gràfics 3D en l'API. Per últim, en 1999, van llançar al mercat el producte més important de la dècada, la GeForce 256 (veure Figura 2.6). Com hem dit abans, aquesta targeta gràfica fou la primera en la qual es començà a utilitzar el terme GPU. Aquest llançament va proporcionar a l'empresa un gran ressò al llarg del món per ser pionera en aquest camp.

En l'any 2004 presentaren la tecnologia *Scalable Link Interface* (SLI) que permet combinar la capacitat de diverses GPU en el mateix ordinador per tal d'incrementar la seua potència gràfica. Després, en 2006, presentaren CUDA, mencionada anteriorment. Amb açò i el llançament de les targetes gràfiques Tesla (2007), que tenien suport per a aquesta API, obriren una nova via als científics i investigadors per poder resoldre els seus problemes més complexos, ja que seria més senzill aprofitar la computació paral·lela de les GPU. En 2008, crearen el primer processador per a telèfons intel·ligents que eren capaços de consumir 30 vegades menys energia que les gràfiques del moment. Per altra banda, en 2008, 2010 i 2012 participaren en la creació de tres supercomputadors, els quals van ser dels que més potència de computació tenien en aquell moment. En 2012 també presentaren la primera GPU virtual, que consisteix en una GPU física ubicada en un servidor. La qual es pot accedir a ella de manera remota, oferint una major facilitat per accedir a ella des de qualsevol dispositiu. Amb açò i el programari GRID[9] traslladaren el processament gràfic al núvol.

[6] https://www.nvidia.com/es-es/

[7] PCI (*Peripheral Component Interconnect*) és un tipus de bus utilitzat per a la interconnexió de la targeta gràfica i la placa base de l'ordinador.

[8] Tècnica per a generar imatges a partir d'un model 2D o 3D.

[9] Programari que permet gestionar l'accés remot a la GPU virtual

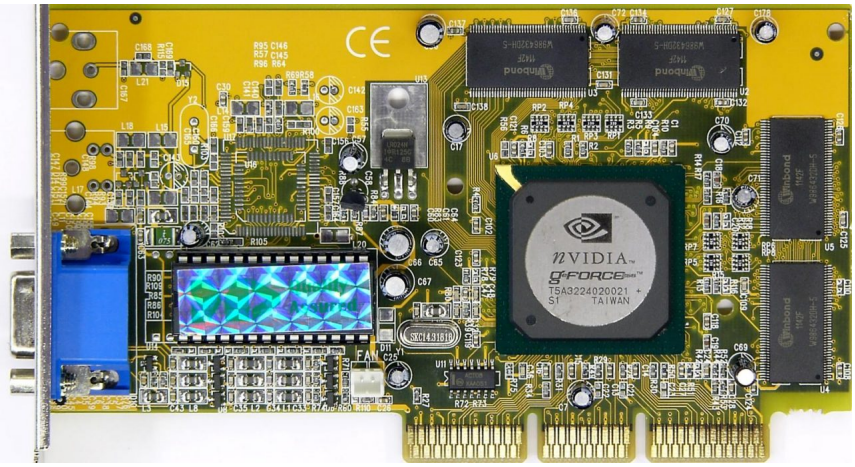

Figura 2.6: Primera targeta gràfica, presentada per l'empresa NVIDIA, on s'instal·là la primera GPU. En la part inferior es pot veure la interconnexió PCI, la qual permet que la targeta es connecte amb la placa base de l'ordinador.

A partir de 2015 i al llarg dels següents anys, l'empresa estatunidenca comença a entrar en el món del *deep learning* i a llançar dispositius específics per a la intel·ligència artificial. El primer que llançaren va ser la Jetson TX1 per a crear màquines intel·ligents i autònomes. Després, al següent any llançaren el primer supercomputador centrat en la IA. En 2017, llançaren la Jetson TX2, predecessora de la Jetson TX1 i que permetia tindre potents robots, drons i càmeres intel·ligents, gràcies a la seua xicoteta mida. Així mateix, en 2018 llançaren diferents programaris per ajudar al desenvolupament dels dispositius mèdics, vehicles autònoms i impulsar la ciència de dades i l'aprenentatge profund. A banda, també llançaren la NVIDIA Jetson AGX Xavier. Per últim, en 2019 presentaren NVIDIA Jetson Nano i NVIDIA Jetson Xavier NX, dos dispositius d'alt rendiment i baixa potència per a aplicacions *Internet of things* (IoT).

Paral·lelament a tot l'esmentat fins ara, NVIDIA ha anat millorant les seues targetes gràfiques modificant la microarquitectura de les seues GPU. En total hi ha uns catorze models, però tan sols ens centrarem en els quatre últims que han presentat, és a dir, des de 2016 fins a l'actualitat. Aquests són:

- Pascal[10] (2016). Amb aquesta arquitectura l'empresa introduí com a novetat en les seues GPU una millora en l'eficiència energètica i una major potència de càlcul respecte a les seues antecessores. Amb aquesta es van poder convertir els ordinadors en superordinadors.

- Volta[11] (2017). NVIDIA se centra en la IA per a crear aquesta arquitectura on l'han dotat de més nuclis per poder aconseguir 125 TFLOPs per segon. A més, compta amb la tecnologia NVLINK que permet una interconnexió d'alta velocitat, sent idònia per a processar dades massives o aplicacions que requerisquen una alta capacitat de còmput.

- Turing[12] (2018). Es posa el focus en la representació gràfica amb nuclis específics per al traçat de rajos en temps real i permetre crear simulacions amb el suport de CUDA.

[10] https://www.nvidia.com/es-es/data-center/pascal-gpu-architecture/
[11] https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/
[12] https://www.nvidia.com/en-us/design-visualization/technologies/turing-architecture/

- Ampere[13] (2020). L'últim llançament de l'empresa on presenta diverses millores respecte a les antigues generacions. Per exemple, un augment de la capacitat de computació i poder tindre una GPU multiinstància, és a dir, una GPU que es pot dividir en diverses instàncies, açò permet que les tasques treballen de manera aïllada.

2.1.3. Característiques NVIDIA Jetson AGX Xavier

En 2015 presentaren el primer dispositiu de la família Jetson, però fou en l'any 2018 quan llançaren l'AGX Xavier [24]. La saga Jetson es va llançar amb l'objectiu d'impulsar la visió artificial, la robòtica i la intel·ligència artificial i que els seus dispositius tingueren un baix consum energètic. L'AGX Xavier està dissenyada per afrontar els reptes que ofereix la intel·ligència artificial i està formada per diferents components, com la CPU, la GPU, la memòria RAM, etc. Les característiques més destacables són:

- Unitat Central de Processament:
  - Basat en la microarquitectura NVIDIA Carmel, la qual està inspirada en els xips de l'empresa ARM, exactament el xip ARM-v8.2. Aquests nuclis incorporen la tècnica *Single Instruction, Multiple Data* (SIMD), emprada per a aconseguir paral·lelisme a nivell de dades. També permet fer operacions amb vectors de nombres reals de simple, doble i mitja precisió. Aquesta CPU fou la primera on es podien fer operacions amb nombres reals de mitja precisió, ja que en les versions anteriors només es podia utilitzar per a emmagatzematge.
  - La freqüència màxima de rellotge que presenten aquests nuclis és de 2,26 GHz cada un. Açò significa que es fan $2,26 \times 10^9$ cicles per segon, en cada cicle canvia el flux de corrent dels transistors. Dit d'una altra manera, a cada cicle la CPU fa un «pas» per fer una operació. Per tant, a major freqüència més operacions per segon és capaç de fer la CPU.
  - Format per huit nuclis que es distribueixen de dos en dos, creant quatre nuclis dúplexs. Cada un té accés a una memòria cau L1[14] de 64 KB, després cada nucli dúplex té una memòria L2[15] de 2 MB. Per últim, els huit cores tenen accés a una memòria cau compartida L3[16] de 4 MB (veure Figura 2.7).
- Unitat de Processament Gràfic amb microarquitectura Volta:
  - 512 nuclis CUDA o *CUDA Cores*, processadors paral·lels que s'encarreguen de processar totes les dades que entren en la GPU. Es podria dir que és com un nucli de la CPU, però menys complex. Igual que la CPU, aquest té unitats ALU, però també unitats *Floating Point Unit* (FPU), els quals fan la mateixa funció que les ALU, però amb nombres de coma flotant.
  - 64 nuclis Tensor o *Tensor Cores*. Són unitats on la seua principal funció és realitzar multiplicacions de matrius de nombres reals de mitja precisió i al resultat es suma una matriu amb el mateix tipus de dada per obtenir com a solució

[13] https://www.nvidia.com/es-es/data-center/ampere-architecture/

[14] La memòria L1 és la més ràpida i més propera als nuclis, la qual emmagatzema les dades i les instruccions que són utilitzades immediatament per la CPU. Aquesta sol dividir-se en dos, una part per a les dades i l'altra per a les instruccions. Cal fer notar que aquesta memòria té un 0,9 ns de latència.

[15] La memòria L2 és un poc més lenta que l'L1 (2,8 ns de latència), però té una major capacitat d'emmagatzematge. Aquesta guarda les instruccions i les dades que seran utilitzades pròximament per la CPU.

[16] La memòria L3 és la més lenta de les tres, ja que té una latència de 11 ns. No obstant això, també és la que major capacitat de memòria té.

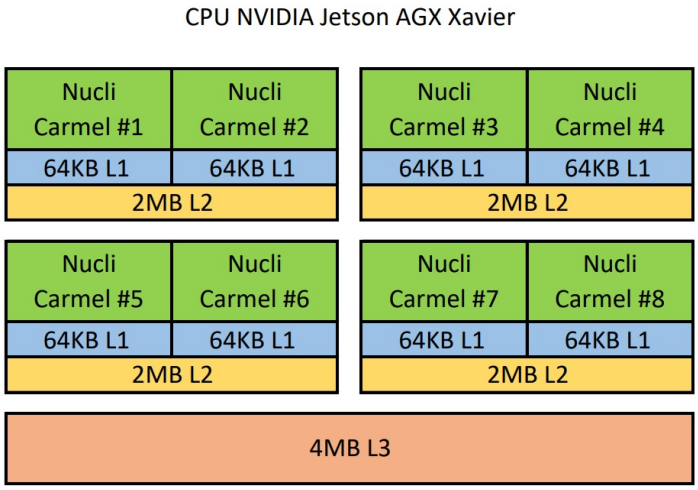

Figura 2.7: Diagrama del disseny de la CPU instal·lada en la Jetson AGX Xavier. Amb aquesta il·lustració es pretén mostrar els components que la formen. Com es pot observar cada nucli té la seua pròpia memòria cau L1, hi han quatre memòries L2 compartides cada una per dos nuclis i l'L3 és compartida pels huit nuclis.

una matriu de nombres reals de simple precisió o mitja precisió (veure Figura 2.8). També té suport per a fer les operacions amb enters. Aquests nuclis es crearen amb l'objectiu d'accelerar l'entrenament de xarxes neuronals. Per últim, cal destacar que aquests són els primers nuclis que va crear l'empresa i la microarquitectura Volta és la primera que els va utilitzar.

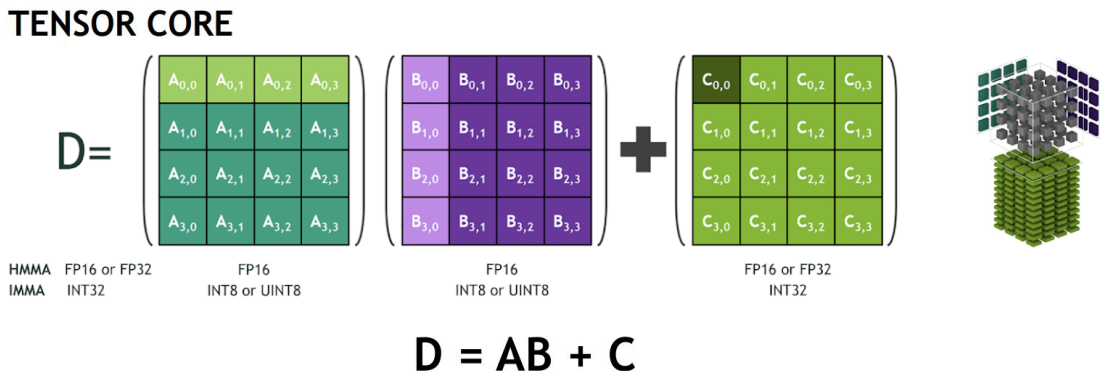

Figura 2.8: Operació que realitzen els *Tensor Core*. La primera matriu és la matriu A, que pot ser de tipus FP16 (nombre real de mitja precisió), INT8 (enter de 8 bits) o UINT8 (enter sense signe de 8 bits). La segona matriu és la matriu B, que pot ser del mateix tipus de dades que l'A. Després, està la C que pot ser de tipus FP16, FP32 (nombre real de simple precisió) o INT32 (enter de 32 bits). Per últim, la matriu D, que és del mateix tipus de dada que la C, emmagatzemarà els valors resultants de l'operació.

- Una freqüència màxima de rellotge d'1,37 GHz per nucli.
- Els nuclis CUDA i Tensor s'agrupen en 8 *Streaming Multiprocessors* (SM) on cada un està format per 64 *CUDA Cores* i 8 *Tensor Cores*. També compta amb una memòria cau L1 de 128 KB i una memòria cau L2 de 512 KB compartida per totes les SM (veure Figura 2.9). Amb açò, és capaç d'aconseguir 11 TFLOPS (operacions en coma flotant per segon) o 22 TOPS (operacions per segon).
- Cada SM ens permet tindre fins a 1024 fils d'execució, és a dir, es pot tindre 1024 tasques simultànies per SM.

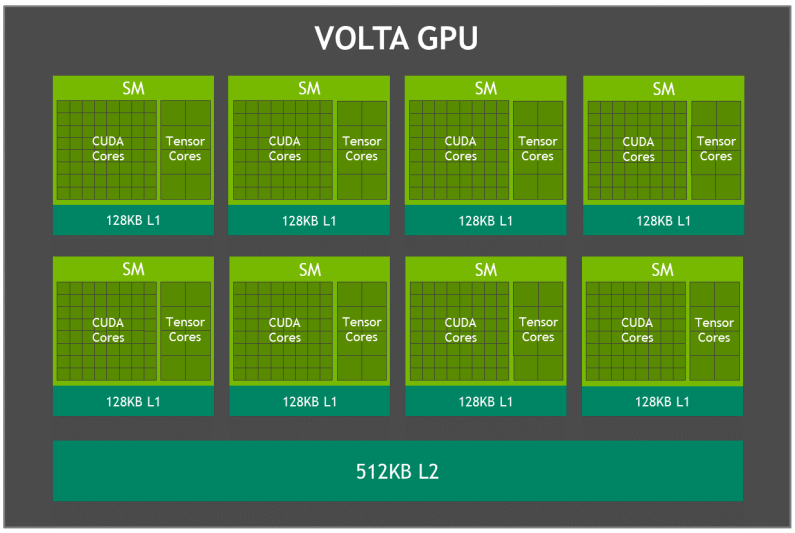

Figura 2.9: Diagrama del disseny de la GPU instal·lada en la Jetson AGX Xavier. Amb aquesta il·lustració es pretén mostrar els components que la formen i il·lustrar la distribució dels nuclis CUDA i Tensor. A més, es pot veure com cada SM té la seua pròpia memòria L1, compartida per tots els nuclis que formen cada SM. També una memòria L2 compartida per tots els SM.

- 2 *Deep Learning Accelerator (DLA)* o Acceleradors d'aprenentatge profund. Són dos motors que s'encarreguen de la inferència en les xarxes neuronals convolucionals per a millorar l'eficiència energètica i ajudar a reduir la quantitat de treball de la GPU. Per tant, permet executar xarxes més complexes i tasques dinàmiques. A més, té un rendiment de 5 TOPS d'enters o 2,5 TFLOPS de nombres reals de mitja precisió per nucli.

- Una memòria RAM[17] de 16 GB LPDDR4x[18], amb una amplada de banda de 137 GB/s, fet que li permet obtenir les dades amb poca latència, i com a conseqüència executar les tasques ràpidament.

- Diferents modes de consum on l'usuari pot escollir entre 10 W, 15 W, 30 W i EDP, el qual permet obtindre el màxim rendiment del dispositiu sense cap limitació de potència. Depenent del mode establit es modifica el nombre de nuclis CPU que s'utilitzen, la màxima freqüència de la CPU, la GPU, les DLA i la memòria. Aquests modes permeten adaptar el dispositiu a la tasca que volem que realitze, per exemple, podem posar el mode de 10 W per a utilitzar el dispositiu en un robot menut o el mode de 30 W o EDP, que permet obtenir un major rendiment, però més consum elèctric, per aplicacions de *Deep Learning*.

- Utilitza el sistema operatiu de codi obert Linux, per oferir una major disponibilitat i facilitat als desenvolupadors. A banda, també té instal·lada l'API CUDA, la biblioteca cuBLAS i cuDNN, TensorRT per a traure el màxim rendiment als nuclis Tensor; OpenGL, etc.

Una vegada descrites les tecnologies aplicades i els components més importants de l'AGX Xavier, cal recalcar que, respecte a les generacions anteriors, aquest dispositiu presenta millores quantitatives [27]. Per exemple, una memòria L1 i L2 molt més ràpida i la incorporació dels DLA que ha permés obtenir una major productivitat . A més, la

[17] La memòria *Random Access Memory* (RAM) és una unitat on s'emmagatzemen les dades de forma temporal i s'utilitza com a memòria de treball per al programari de l'ordinador per la seua poca latència i rapidesa.

[18] Una memòria de tipus LPDDR4x és una memòria destinada a dispositius mòbils, la qual permet un alt rendiment amb un consum baix d'energia.

incorporació de la interconnexió PCIe o PCI *Express* de quarta generació ofereix una major velocitat de comunicació entre els diferents components. Amb aquestes millores NVIDIA ha demostrat que ha posat el focus en l'aprenentatge profund i ha aconseguit millorar els seus dispositius per ajudar a assolir els reptes actuals de la intel·ligència artificial.

2.2 Xarxes neuronals

Les xarxes neuronals estan en auge actualment per tots els assoliments que han aconseguit, com per exemple generar peus de fotos automàticament o el reconeixement de la parla dels assistents de veu. Aquest increment de la popularitat també es deu als avanços de la GPU i el fet que les operacions tan costoses com les que s'han de realitzar en les xarxes es puga fer en qüestió de segons. Amb aquesta secció es pretén posar en context i explicar què són.

Aquestes estan inspirades en les xarxes neuronals del cervell humà i imiten la comunicació entre elles. Una xarxa neuronal artificial està formada per una capa d'entrada, una d'eixida i després poden tindre una o més capes ocultes. La capa d'entrada accepta les dades, normalment amb una estructura de vector, les quals envia a la següent capa que les processarà, les capes ocultes. Aquestes tenen la característica que tan sols poden rebre i enviar la informació a altres neurones. Per últim, està la capa d'eixida que retorna les dades processades per les capes ocultes (veure [7], pàgina 36).

Les capes estan compostes per neurones i per explicar què són ens hem de remuntar al 1943 quan Warren McCulloch i Walter Pitts presentaren el concepte de neurona artificial i un model informàtic basat en la lògica booleana. A partir d'aquest treball Frank Rosenblatt creà el perceptró: a través de diverses entrades binàries produïa una eixida binària. Per a calcular l'eixida, introduí pesos a les entrades de les neurones. El pes establia l'impacte que té l'entrada en la neurona: com més gran era el pes, més contribuïa a l'eixida correcta. Amb aquest mètode la xarxa neuronal era capaç d'entrenar i aprendre per ella sola, perquè anava modificant els pesos per poder augmentar la seua precisió i trobar el resultat desitjat (veure Figura 2.10). Aquests entrenaments eren exemples perquè la xarxa practicara la identificació o classificació de les dades.

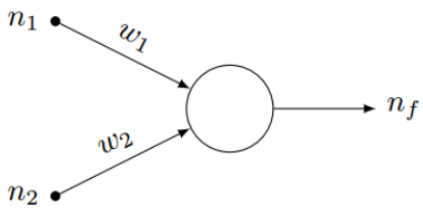

Figura 2.10: Aquesta imatge il·lustra les entrades de la neurona (n_i), els pesos de cada entrada (w_i) i l'eixida (n_f).

Al llarg dels anys es va demostrar que les neurones basades en perceptró no eren tan eficients, ja que quan es modificava un pes de la xarxa, podia canviar completament el resultat que s'obtenia i feia impossible assolir uns bons resultats d'identificació. Com a conseqüència, es creà un nou tipus de neurones, les neurones *feedforward* o *Multi-layer Perceptrons* (MLP), perquè en comptes de tindre valors d'entrada binaris, tingueren un número entre el 0 i l'1. Amb açò, si es feia un xicotet canvi en algun dels pesos, tan sols produïa un xicotet canvi en l'eixida facilitant l'aprenentatge de la xarxa neuronal. El funcionament de les neurones ocultes és la realització d'un sumatori amb les dades que rep, el sumatori és el següent [16]:

$$\text{op} = \sum_{i=1}^{m} w_i * n_i + \text{bias}$$

On «w_i» es el pes, «n_i» es l'entrada i «bias» es el llindar marcat.

Les neurones d'eixida obtindran les dades processades i realitzaran la següent comparació per tornar un resultat o altre.

$$\text{Eixida} = \begin{cases} 1 \text{ si op} \geq 0 \\ 0 \text{ si op} < 0 \end{cases}$$

A banda de les neurones de tipus perceptró i les MLP, també estan la xarxa neuronal convolucional (*Convolutional Neural Networks*, CNN). Aquestes són similars a les xarxes *feedforward*, però s'utilitzen per al reconeixement d'imatges, patrons, etc. Açò ho aconsegueixen aprofitant les multiplicacions matricials per a identificar patrons. Per últim, també estan les xarxes neuronals recurrents (*Recurrent neural networks*, RNN) que tenen bucles de retroalimentació, una tècnica molt útil per a realitzar prediccions sobre resultats futurs.

2.2.1. Funcions d'activació

Les funcions d'activació o de transferència serveixen perquè quan una eixida sobrepasse el llindar marcat es modifique el valor per un més manejable (veure Figura 2.11). Amb aquesta tècnica es poden filtrar les eixides i millorar l'aprenentatge automàtic. Existeixen distints tipus de funcions d'activació, però ens centrarem en les més importants (veure [7], pàgines 40 – 46).

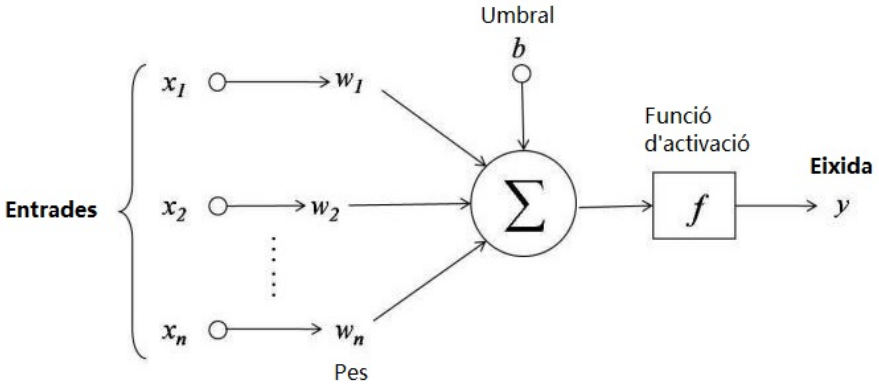

Figura 2.11: La neurona rep les dades d'entrada, els pesos i el llindar, amb aquestes dades realitza el sumatori descrit anteriorment (op). Després, s'aplica una funció d'activació on depenent del tipus es realitza una opció o una altra.

Funció sigmoide

La funció sigmoide és una funció molt comuna per a quan només es vol generar nombres positius. Aquesta funció descriu la progressió temporal que mostren els sistemes d'entrenament.

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

Com s'observa a la Figura 2.12, l'eixida sofreix un gran increment quan els valors d'entrada són 0 o estan prop d'aquest valor. Aquesta gràfica es basa en la funció anterior on els resultats estaran entre 0 i 1.

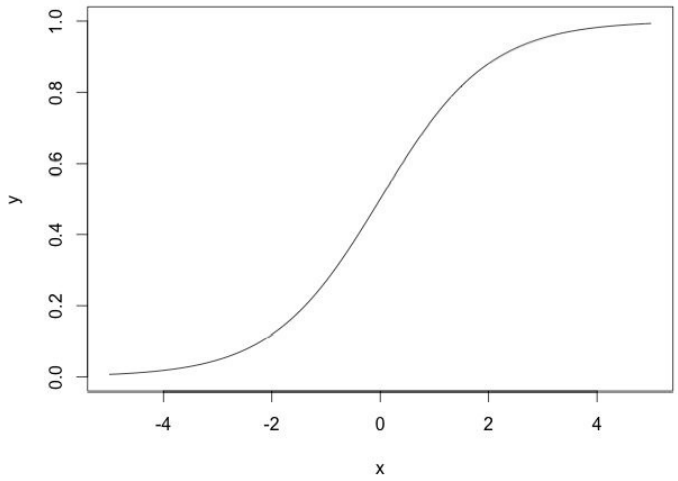

Figura 2.12: En la gràfica tenim sobre l'eix d'abscisses el valor d'entrada i en l'eix d'ordenades el valor d'eixida. Es pot observar com si el número d'entrada és negatiu es traduirà com un valor pròxim a 0 i si és positiu a un valor pròxim a 1.

Funció ReLU

La *Rectified Linear Unit* (ReLU) és una de les funcions d'activació més populars pels bons resultats que aporta, ja que no se satura a 0 o 1, com la sigmoide, sinó que s'obté el valor màxim, sempre que siga positiu; en cas de no ser-ho el valor d'eixida serà 0. La funció per a calcular-ho és:

$$f(x) = \max(0, x)$$

La gràfica que obtenim a partir d'aquesta funció es pot veure en la Figura 2.13.

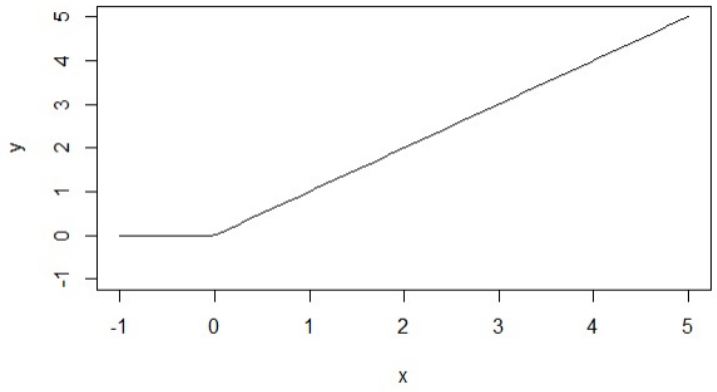

Figura 2.13: El valor d'eixida (y) tan sols se satura a 0 si el valor és negatiu; en cas que siga positiu el valor d'eixida no se saturarà.

Funció *softmax*

La funció *softmax* sol utilitzar-se per a les xarxes neuronals de classificació, ja que la neurona d'eixida seleccionarà l'entrada amb el valor més elevat de la seua classe. En aquesta eixida es mostrarà un percentatge, el qual ajudarà a trobar la classe que més probabilitats té de ser la correcta. La funció que ho calcula és:

$$\phi_i = \frac{e^{z_i}}{\Sigma_{j \in \text{grup}}\, e^{z_j}}$$

On ϕ és la neurona d'eixida, i és el índex d'aquesta, j representa els índex de totes les neurones del grup o nivell i la variable z és l'eixida de la neurona.

2.3 HELENNA

HELENNA és una plataforma de programari per al desenvolupament de processos d'entrenament i inferència de xarxes neuronals. HELENNA és l'acrònim d'*HEterogeneous LEarning Neural Networks Application*. Com a tal, té com a objectiu l'ús d'arquitectures heterogènies de còmput.

Encara que existeixen múltiples solucions per a l'entrenament i inferència de xarxes neuronals, HELENNA s'ha desenvolupat per a explorar nous mètodes d'entrenament i noves xarxes neuronals en el context de projectes d'investigació. L'ús d'una plataforma desenvolupada exclusivament pel grup d'investigació GAP permet un major control sobre els processos d'entrenament i inferència, i per tant, una millor adequació a l'estudi i anàlisi de processos alternatius que milloren el seu rendiment.

HELENNA està desenvolupat en els llenguatges de programació C i C++ (principalment en C) i té una estructura de codi similar a la programació en objectes encara que no els utilitza. En concret, HELENNA permet la definició de capes de xarxes neuronals aïllant el codi de la capa en un únic fitxer i implementant diferents funcions bàsiques per al suport de l'entrenament i inferència de la capa. Des d'aquest punt de vista, crear el suport per a una nova capa comporta la creació d'un nou fitxer i la instanciació de les funcions de la capa a suportar. Cada capa suportada en HELENNA ha d'implementar cinc funcions bàsiques:

- Funció *parse*: interpreta els paràmetres de definició de la capa i inicialitza les seues estructures.
- Funció *allocate*: amb aquesta funció la capa crea els *buffers* (tensors) adequats per al suport de la capa.
- Funció *initialize*: la funció inicialitza els *buffers* (tensors) de la capa.
- Funció *forward*: funció que realitza el procés de *forward* (inferència) de la capa.
- Funció *backpropagation*: es calculen els gradients de la capa per a la seua posterior aplicació sobre els pesos.
- Funció *propagate_error*: funció que calcula l'error en l'entrada de la capa a partir de l'error en l'eixida de la capa.

Les últimes tres funcions enumerades anteriorment serveixen per a realitzar el procés d'entrenament de la capa una vegada inserit en una xarxa neuronal.

Un altre aspecte d'HELENNA és la gestió de memòria i l'abstracció de la ubicació dels *buffers* de memòria que s'utilitzen en un procés d'entrenament o inferència. De fet, els tensors (implementats amb *buffers*) es creen de manera dinàmica en HELENNA i se situen en el dispositiu objectiu seleccionat (CPU, GPU, FPGA, etc.). L'usuari no és conscient del lloc on estan situats en el moment de la instanciació i en l'ús dels *buffers*. HELENNA realitza internament les transferències necessàries per a una correcta utilització dels *buffers*/tensors. Cada tensor creat en HELENNA s'identifica amb un identificador únic i cada acció sobre el tensor utilitza exclusivament l'identificador.

HELENNA permet l'entrenament en CPU i en GPU amb diferents optimitzacions, principalment en el suport de biblioteques de càlcul matricial. En concret, HELENNA permet els mòduls següents:

- CPU. S'utilitza la CPU, tots els seus nuclis, en l'entrenament o inferència. No s'utilitza cap biblioteca de càlcul matricial. Aquest dispositiu és el bàsic.
- MKL. S'executa amb la CPU junt amb la biblioteca de càlcul matricial MKL d'Intel. Aquesta biblioteca funciona exclusivament amb processadors Intel. Amb aquest dispositiu s'obté el màxim rendiment utilitzant la CPU.
- Cublas utilitza la GPU de NVIDIA. En concret, fa servir la biblioteca cuBLAS de NVIDIA per al càlcul matricial.
- CuDNN. Recentment, NVIDIA ofereix CuDNN com a una biblioteca per a l'entrenament i inferència de xarxes neuronals. Aquesta funciona sobre les GPU de NVIDIA. Aquest dispositiu fa servir la biblioteca per a millorar significativament les prestacions.
- OpenCL-FPGA. Amb aquest mòdul s'utilitzen sistemes basats en FPGA per al procés d'inferència. En concret, aquest permet utilitzar mòduls generats per a FPGA específics com a convolucions i funcions d'activació.

A banda dels mòduls descrits anteriorment, s'ha implementat un de nou on s'utilitza la biblioteca cuBLAS i els nombres reals de mitja precisió. Aquest està basat en la versió de Cublas, el qual està implementat amb nombres reals de precisió simple. L'objectiu del nou mòdul és explotar els nombres de precisió reduïda en l'entrenament i inferència de xarxes neuronals per obtenir una major eficiència i realitzar entrenaments en menor temps. Aquest nou mòdul ha sigut implementat per poder ser utilitzat en la majoria de les capes de xarxes neuronals disponibles en HELENNA.

Actualment HELENNA suporta una varietat de capes de xarxes neuronals. En la Taula 2.1 es llisten les capes suportades, així com alguns detalls de cadascuna d'elles.

Capa	**Paràmetres**	**Descripció**
Fully_connected	Nombre de paràmetres	Capa densa, també coneguda com *MultiLayer Perceptron* (MLP)
Convolution	Grandària (KH, KW) de filtre, Grandària (SH, SW) de *stride*, Grandària (PH, PW) de *padding*	Capa convolucional
Batch Normalization	Gamma i Beta	Capa de normalització
Activation: ReLU, Sigmoide, etc.		Funcions d'activació
Softmax		
Dropout	llindar	Capa de normalització

Taula 2.1: Capes suportades per HELENNA

A més, se suporten diferents formats de bases de dades així com ara bases de dades molt utilitzades en el context de les xarxes neuronals. Aquestes són:

- MNIST[19] (*Mixed National Institute of Standards and Technology database*) és una base de dades composta per imatges de dígits escrits a mà. Aquesta té al voltant de

[19] http://yann.lecun.com/exdb/mnist/

60.000 imatges i és de les més utilitzades, sobretot a l'inici de la implementació d'una xarxa, per la seua simplicitat. Aquesta base de dades tan sols necessita una xarxa neuronal que siga capaç d'identificar les zones pintades de negre o de blanc, en cas de tenir la imatge invertida, i identifiquen el número escrit (veure Figura 2.14).

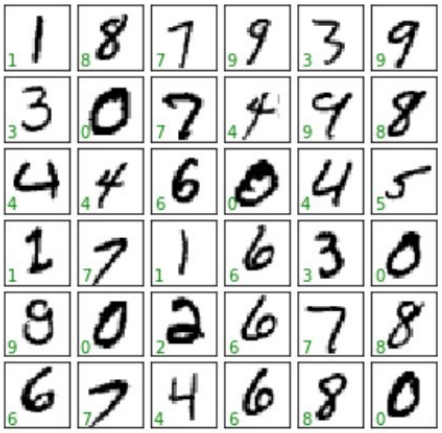

Figura 2.14: Xicotet exemple de les imatges incloses en la base de dades.

- Cifar-10[20] (*Canadian Institute For Advanced Research*) és una col·lecció de 60.000 imatges amb 10 classes diferents, és a dir, hi ha 6.000 imatges d'avions, cotxes, aus, gats, cérvols, gossos, granotes, cavalls, vaixells i camions. Amb aquestes imatges la xarxa neuronal identifica quin objecte o animal és el que està representat en la fotografia (veure Figura 2.15).

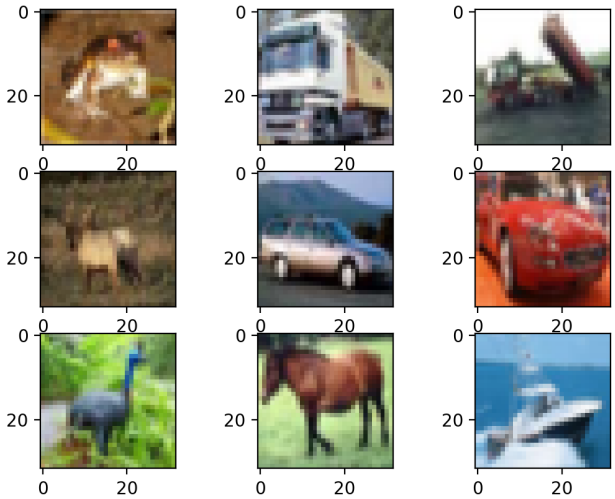

Figura 2.15: Exemple de 9 imatges que formen aquesta base de dades on mostren l'objecte a identificar. Aquestes imatges són de 30x30 píxels, aleshores presenten una resolució reduïda, però suficient per a que es puga identificar el objecte que representa.

- Cifar-100 és una base de dades similar a la Cifar-10, ja que conta amb 60.000 imatges de diferents classes, pero el nombre de classes es significativament major que les que presenta Cifar-10, ja que té 100 classes amb 600 imatges cada una.

[20] https://www.cs.toronto.edu/~kriz/cifar.html

- ImageNet[21] és una base de dades que inclou 1.2 milions d'imatges i 1.000 categories per a ser utilitzada en la investigació del reconeixement d'objectes visuals.

Finalment, HELENNA permet l'entrenament distribuït tant en CPU com en GPU. Diferents estratègies d'entrenament distribuït estan suportades en HELENNA.

2.4 Nombres reals, CUDA i biblioteques

El present treball fi de grau s'ha desenvolupat amb la utilització de l'entorn de programació CUDA, biblioteques, com ara cuBLAS i nombres reals de mitja precisió. Aquesta secció té l'objectiu d'explicar detalladament què és aquest tipus de dada, per què s'ha escollit per a implementar-lo en HELENNA i en què consisteix el model de programació CUDA i les biblioteques utilitzades.

2.4.1. Nombres reals

Els nombres reals són tots els nombres que estan entre infinit i menys infinit, és a dir, en aquest conjunt entren els nombres racionals (positius, negatius i el 0), nombres irracionals i els transcendents, com π. Els nombres reals es poden representar en decimal i en binari ([26], pàgina 7).

Per processar aquests nombres en els computadors hi han diferents tipus de representació, però en aquest treball tan sols ens centrem en la representació en coma flotant. La representació en coma flotant està basada en la notació científica, per exemple 5.238×10^{-3}, i permet reproduir una quantitat limitada de nombres reals. Per a representar els nombres es treballarà amb els dígits més significatius, de manera que el nombre real no es podrà crear amb una precisió total. Quan la part fraccionària d'un nombre real és massa gran per ser representada en un computador, aleshores s'ha de fer algun tipus d'ajustament, la qual cosa produeix una pèrdua d'exactitud. Hi ha distintes maneres de fer aquests ajustos i reben genèricament el nom de tècniques d'arredoniment. Aquest arredoniment pot ser de molts tipus:

- Al més pròxim: s'arredoneix al nombre al valor més pròxim, és a dir, si es té 4.537, el número resultant de l'arredoniment és 4.54.
- A zero: en aquest cas es trunca el valor. Seguint l'exemple anterior de 4.537 s'arredoneix a 4.53.
- A més infinit: s'arredoneix els valors aproximant-se a més infinit, és a dir, si el nombre a arredonir està més prop de l'infinit, aquest augmenta. Per exemple, 4.537 s'arredoneix a 4.54. Si ocorre el contrari, el valor es trunca, 4.533 s'arredoneix a 4.53.
- A menys infinit: en aquest cas els valors a arrodonir es fa aproximant-se a menys infinit. Si el nombre està més prop a menys infinit, s'arredoneix, si no es trunca. Per exemple, 4.537 s'arredoneix a 4.53 i -4.537 s'arredoneix a -4.54.

En els computadors existeixen diferents tipus de nombres en coma flotant. Aquests es divideixen depenent de la quantitat de bits que tenen per representar els valors. Dit d'una altra forma, la quantitat de valors que poden representar (veure Figura 2.16). Aquests són:

[21] https://www.image-net.org/index.php

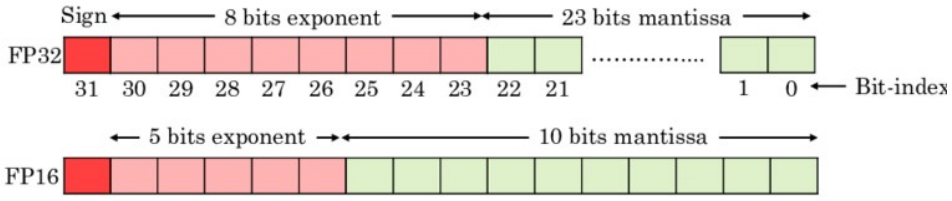

Figura 2.16: Diagrama del format dels nombres en coma flotant de simple (FP32) i mitja (FP16) precisió. Com es pot observar els dos tenen 1 bit de signe, el qual indica si el nombre està en positiu o en negatiu, s = 1 està en negatiu i si s = 0 està en positiu. A banda, hi ha 8 o 5 bits d'exponent depenent del tipus i 32 bits de mantissa per a FP32 o 10 bits per a FP16.

- Format de doble precisió: aquest format té una mida de 64 bits.
- Format de simple precisió: format amb 32 bits.
- Format de mitja precisió: format amb una mida de 16 bits.

Els de doble precisió són els que més números poden representar i també el que major precisió dels tres té. Però en aquest treball tan sols ens centrem en els altres dos, perquè el seu ús en les xarxes neuronals les faria molt costoses. A banda, no és necessària una precisió tan elevada per a l'entrenament de les xarxes. Com a conseqüència, aquesta precisió quasi no s'utilitza en la inferència i l'entrenament.

Segons l'estàndard IEEE 754-2008 els nombres en coma flotant es classifiquen en cinc classes de codificació (veure [10], pàgines 182 – 192). Aquestes són:

- Els valors normals: són tots aquells que es poden obtenir amb la següent equació:

$$F = (-1)^s \times 2^{(\text{exp}-\text{bias})} \times 1.f$$

  On «s» és el bit de signe, «exp» és l'exponent, «bias» és la constant, que pot ser 127 en els nombres de simple precisió o 15 en els de mitja precisió. Per últim, $1.f$ és un valor entre el rang [1.0, 2.0) i es refereix a la mantissa o *fraction*. Un exemple per a representar un valor en simple precisió a través de l'equació és:

$$6.5 = -1^0 \times 2^2 \times 1.625$$

  Com que és positiu $s = 0$, $exp = 2$, ja que $2^2 = 4$ i $2^3 = 8$. S'agafa el valor més pròxim al valor que volem representar, però també ha de ser el més menut, per això s'agafa el 2^2. Per calcular la mantissa, en aquest cas 1.625, es fa la següent divisió: $6.5/2^2$. Una vegada trets aquests valors es calcula el seu valor en binari, on 1.625 és 1.101 i l'exponent esbiaixat és 129, en binari és 1000 0001 (veure la Figura 2.17).

S	Exponent								Fraction																							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4				0				D			0				0				0				0				0				

Figura 2.17: Representació gràfica de l'exemple exposat on es distribueixen els bits al llarg dels 32 bits que té la coma flotant de simple precisió. També, s'ha convertit a hexadecimal per ajudar a la seua lectura i comparar-lo amb la representació del valor en coma flotant de mitja precisió.

- Els valors subnormals: aquests valors són aquells que no són 0, però són massa menuts per a representar-se en un rang normal. Aleshores es representen amb una codificació fora del que és normal. Aquests valors satisfan la següent equació:

$$F = (-1)^s \times 2^{-126} \times 0.f$$

  L'exponent -126 correspon al mínim exponent dels valors en coma flotant de simple precisió, en cas de calcular-ho per a mitja precisió el valor és -14. A més, $0.f$ és un valor entre el [0.0, 1.0). Aquest es calcula igual que el $1.f$, vist anteriorment.

- Zeros: el 0 en la representació IEE 754 pot ser positiu o negatiu. Açò implica que en determinades operacions s'obtinga un signe diferent del que es deuria. Per exemple, $1/0 = \infty$, però si 0 és negatiu l'operació dona $-\infty$.

- *Not a Numbers* (NaN): els valors NaN es codifiquen posant tots els bits de l'exponent i la mantissa a 1. Aquest valor pot aparéixer per l'execució d'una operació invàlida.

- Infinits: aquest es codifica tant en simple com en mitja precisió posant tots els bits de l'exponent a 1 i tots els de la mantissa a 0. Per altra banda, el bit de signe mostra si és $+\infty$ o $-\infty$. També cal recalcar que quan s'obté o s'intenta representar un valor fora del rang de representació, tant en simple com en mitja precisió, aquests es codifiquen en $+\infty$ si el valor és positiu, o $-\infty$ si el valor és negatiu.

Els nombres reals de precisió reduïda són com els nombres reals de simple precisió, però amb la meitat de precisió. Per tant, la dada té una mida de 16 bits o 2 bytes en memòria. Açò significa que es poden representar des de -64.504 fins a 64.504, la qual cosa té uns avantatges, però també uns inconvenients. En tenir una menor mida que el FP32 permet que les operacions matemàtiques puguen tindre un cost temporal menor. Per exemple, en fer una multiplicació de matrius, es pot realitzar la multiplicació amb menys temps que amb FP32. Al contrari, el fet que siga més menut té com a conseqüència una pèrdua de precisió. Aquest fet pot implicar que en algunes aplicacions on la precisió és molt important, no siga útil fer ús d'ella. Però també s'ha d'indicar que moltes voltes la pèrdua de precisió no és molt notòria; aleshores en aplicacions com les xarxes neuronals resulten de molta utilitat pel gran rendiment que aporta [25].

Per veure un exemple de la diferent representació dels valors entre els nombres en coma flotant de mitja i simple precisió, es posa com a exemple la representació en mitja precisió del valor 6.5 que s'ha vist abans. L'equació per obtenir-la és la mateixa:

$$6.5 = -1^0 \times 2^2 \times 1.625$$

L'únic que canvia a l'hora de representar-lo són els bits, ja que la mantissa seguirà sent la mateixa, però l'exponent esbiaixat és 17, en binari és 0001 0001. Per tant, si ho representem en hexadecimal tindrem el valor 0x40D00000 en coma flotant de simple precisió i 0x4680 en coma flotant de mitja precisió (veure Figura 2.18).

Per últim, cal fer notar que els nombres reals de simple precisió (veure [35], pàgina 8 – 13), s'anomenen *float* en la majoria de llenguatges de programació. Els de mitja precisió depenent del llenguatge de programació que s'utilitze es poden anomenar d'una forma distinta. Per exemple, en el llenguatge C s'anomenen *__fp16* i en el llenguatge C++ i CUDA s'anomenen *__half*.

2.4.2. CUDA

CUDA (*Compute Unified Device Architecture*), creada per NVIDIA, és un entorn de treball per a desenvolupar algoritmes amb paral·lelització de tasques en les GPU de propòsit

S	Exponent					Fraction									
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	1	0	0	0	0	0	0	0
4				6				8				0			

Figura 2.18: Diagrama on es mostra com es representa en bits el valor 6.5 en la codificació en coma flotant de mitja precisió. També s'ha representat en hexadecimal per comparar-lo amb la codificació en coma flotant de simple precisió d'una forma més senzilla.

general. A més, té una API i un llenguatge de programació basat en el llenguatge C/C++. Actualment, és d'estàndard obert i l'empresa proporciona el programari i les biblioteques necessàries per a desenvolupar les aplicacions. Aquest *framework* tan sols es pot utilitzar en les GPU de NVIDIA, perquè aquest va ser dissenyat per poder aprofitar tot el potencial dels *CUDA Cores*, ja que permet al programador indicar com vol que es dividisca la tasca d'una manera molt senzilla i CUDA s'encarrega de fer-ho.

CUDA no és l'única plataforma de programació per a sistemes dedicats a la computació d'alt rendiment ([2], pàgines 16 i 17). També està la plataforma OpenCL (*Open Computing Language*). Aquesta està enfocada per a poder executar-se en CPU, GPU i en les matrius de portes lògiques programables en camp o *field-programmable gate array* (FPGA). OpenCL és de codi obert i permet utilitzar-se en quasi tots els dispositius del mercat, ja que té el suport de NVIDIA, AMD, etc. Aquest és molt similar a CUDA, però una mica més complex. Una altra alternativa a CUDA, però en el món de les CPU, seria OpenMP (*Open Multi-Processing*), que permet una paral·lelització del codi que s'executa a la CPU; en aquest treball també s'utilitza per a realitzar unes optimitzacions que s'explicaran en el següent capítol.

Model de programació

A l'hora de programar amb CUDA s'han de tenir en compte diferents aspectes com: on es creen les estructures de dades, com s'indica quin codi s'ha d'executar en la CPU i quin en la GPU o com es divideixen les dades i les tasques en la GPU.

En primer lloc, tant la CPU com la GPU tenen la seua pròpia memòria. Aleshores, totes les dades que es definisquen en un dels dos dispositius no estarà disponible en l'altre, és a dir, si es defineix un vector de tipus *float* en la CPU, també s'haurà de definir en la GPU. Per fer açò, s'han implementat algunes funcions, basades en el llenguatge C. Aquestes es poden veure en la Taula 2.2.

Llenguatge C	**Llenguatge CUDA**	**Funció**
`malloc`	`cudaMalloc`	Assignació de memòria.
`free`	`cudaFree`	Alliberament de la memòria assignada.

Taula 2.2: Funcions d'assignació i alliberament de memòria en llenguatge C i CUDA.

Un exemple de com assignar memòria d'un vector de tipus *void* és:

```
cudaMalloc( (void**) a, (size_t) size);
```

On `a` seria el vector i `size` seria la mida.

D'altra banda, també es pot passar la informació des de la CPU a la GPU o viceversa. Açò permet definir variables en CPU i després passar-les a la GPU per a realitzar els

càlculs pertinents, una vegada la GPU tinga el resultat el torna a passar a la CPU. Per passar la informació s'utilitza la funció `CudaMemcpy`. Un exemple del seu ús és:

- Per a fer el pas de CPU a GPU:

```
cudaMemcpy(gpu_m, cpu_m, size, cudaMemcpyHostToDevice);
```

- Per a fer el pas de GPU a CPU:

```
cudaMemcpy(cpu_m, gpu_m, size, cudaMemcpyDeviceToHost);
```

La variable `gpu_m` és un vector assignat en la GPU i `cpu_m` és un vector assignat en CPU, `size` el nombre d'elements que s'han de copiar i `cudaMemcpyHostToDevice` per indicar que la transferència es fa de CPU a GPU i `cudaMemcpyDeviceToHost` per fer l'operació inversa ([1], pàgina 27).

En segon lloc, s'ha de tenir en compte que CUDA està dissenyat per a sistemes heterogenis. Aleshores, quan s'implementa el codi, s'ha d'indicar que seccions del codi, s'han d'executar en CPU i quines en GPU. Per indicar-ho CUDA ha afegit unes etiquetes que es posen en la capçalera de les funcions per indicar on s'ha d'executar. Aquestes són:

- `__global__`. La funció amb aquesta etiqueta indica que s'ha d'executar en la GPU. Estes funcions podran ser cridades des de CPU i GPU.
- `__device__`. Aquesta indica exactament el mateix que l'anterior, però tan sols es pot cridar des de la GPU.
- `__host__`. Amb aquesta s'indica que el codi ha d'executar-lo la CPU i tan sols es poden cridar des de la CPU.

Igualment, l'única etiqueta realment necessària és `__global__`, perquè qualsevol funció que no tinga aquesta etiqueta la GPU no la considerarà i, per tant, només la realitzarà la CPU ([1], pàgina 38).

Les funcions que tinguen el prefix `__global__` són anomenades *kernel*. Les crides a aquestes necessiten unes variables, les quals s'indiquen entre símbols <<< i >>>, per indicar el nombre de blocs[22] (*blocks*) que volem utilitzar i el nombre de fils[23] (*threads*) que volem que tinga cada bloc (veure Figura 2.19). Amb aquestes variables, CUDA divideix les dades entre tots els fils que van a executar la funció. Un exemple senzill d'una funció *kernel* és:

```
__global__ prova_device(float *a, float b, int c){
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if(tid < n){
        b = a[tid] + a[1];
    }
}
//S'executa en la CPU
void prova_kernel(float a*, float b, int c){
    int numBlock = 2;
    int numThread = 4;
```

[22] Els blocs estan formats per una quantitat definida de fils d'execució. Un conjunt de blocs formen una malla o *grid*

[23] El nombre de fils indica en quants nuclis executaran la funció.

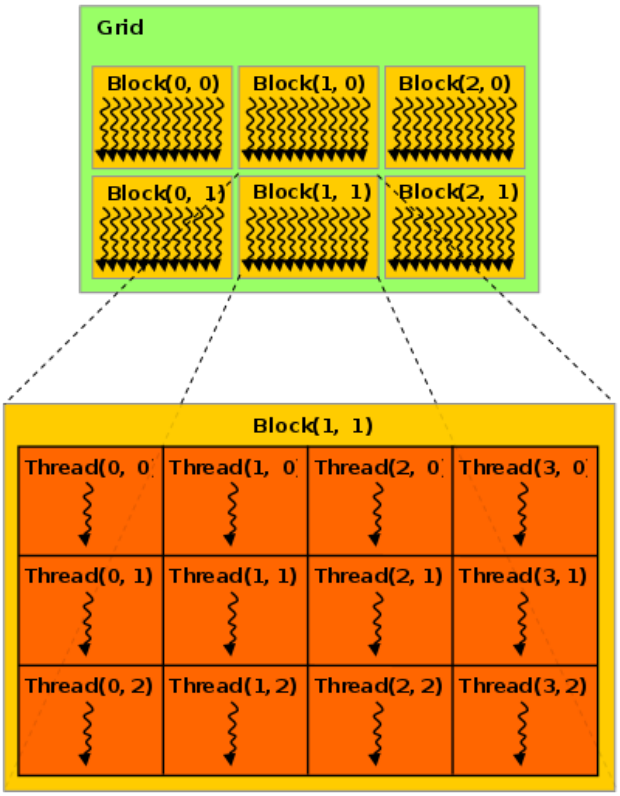

Figura 2.19: Diagrama on s'observa el *grid* format per 6 blocs on cada bloc està format per distints fils d'execució.

```
    cudaDeviceSynchronize();
    prova_device<<<numBlock, numThread>>>(a, b, c);
    cudaDeviceSynchronize();
}
```

En el codi definim el nombre de blocs amb la variable `numBlock` i el nombre de fils amb la variable `numThreads`. Per tant, tindríem dos blocs amb 4 fils, aleshores en aquest cas tan sols tindríem 8 fils d'execució. Així que s'executarà la suma de la funció en 8 nuclis. Cal destacar, que el nombre màxim de fils que es poden posar per bloc són de 1024 des del llançament de la GPU Volta. Tanmateix, també s'ha de tenir en compte que els blocs poden ser de 1D, 2D o 3D; com que es tracta d'un exemple d'explicació s'ha preferit mostrar com seria la creació del model en 1D ([2], pàgina 77).

Després, la variable `tid` és per indicar la posició en memòria de dada a la qual es vol accedir. Per obtenir aquesta posició es fa l'operació amb les seues variables ja definides en CUDA, on cada fil té la variable guardada com a local, és a dir, cada fil obté un valor distint de identificació (veure [33], pàgina 213):

- `blockIdx.x`: retorna el número del bloc al qual correspon el fil.
- `blockDim.x`: retorna el nombre total de fils que té el bloc.
- `threadIdx.x`: retorna l'índex del fil actual.

Totes aquestes variables tornen el valor de la dimensió 1, per accedir, en cas d'haver-ho definit, als valors de la segona dimensió seria el mateix, però canviant la «x» per «y»,

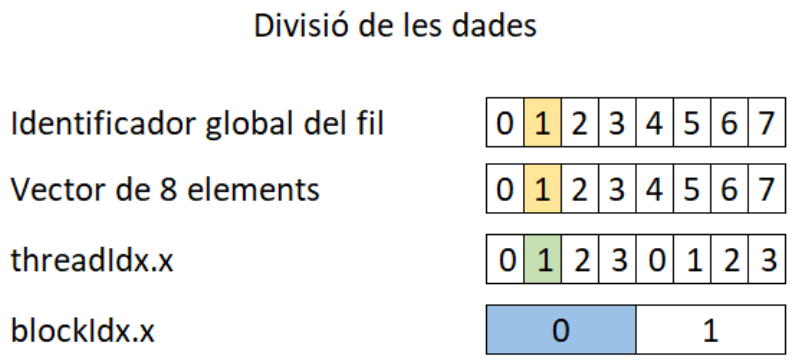

Figura 2.20: Exemple de la divisió de les dades d'un vector de 8 elements entre dos blocs de 4 fils cada un. L'identificador global del fil correspon a la variable `tid` del codi d'exemple. En aquest cas correspon a les posicions del vector.

per exemple `blockIdx.y`. Si volguérem accedir a la tercera dimensió seria canviar-ho per «z».

Les dades que es passen a una funció es divideixen entre els blocs i els fils que s'hagen definit en el *kernel* (veure [1], pàgina 37). Com es pot observar l'exemple de la Figura 2.20, per obtenir el valor de la posició 1 del vector, s'ha de calcular el valor d'identificació global del fil o, en el cas del codi, la variable `tid`. Per tant:

$$tid = blockIdx.x * blockDim.x + threadIdx.x$$
$$tid = 0 * 4 + 1$$
$$tid = 1$$

Per últim, la funció `cudaDeviceSynchronize()` serveix per a sincronitzar la CPU i la GPU, és a dir, amb aquesta crida es bloqueja la CPU fins que la GPU haja acabat la tasca que estiga realitzant. Així assegurem què no es faça una operació amb una variable que hauria de ser tractada per la GPU en primera instància, ja que si la CPU opera sobre la variable abans que la GPU haja fet els càlculs pertinents, totes les operacions posteriors que la necessiten seran errònies.

Gestió de memòria

En CUDA existeixen diferents tipus de memòria (veure [2], de la pàgina 120 – 199) cada una amb les seues propietats i característiques:

- **Memòria compartida**: per a emmagatzemar aquestes dades s'utilitza la memòria cau L1. Les variables que s'assignen en memòria compartida són compartides per tots els fils del mateix bloc. Per tant, s'han de crear mecanismes de sincronització perquè no sorgisca cap incongruència en les dades.
- **Memòria local**: memòria privada que tan sols pot ser accedida pel fil.
- **Memòria global**: es poden utilitzar variables d'aquest tipus tant en CPU com en GPU. Aquest tipus de memòria presenta el avantatge de que no es necessari passar els valors de la memòria de la CPU a la memòria de la GPU o viceversa, ja que es creen en els dos.
- **Memòria constant**: memòria que s'emmagatzema en la cau i és tan sols de lectura.

2.4.3. Biblioteques

cuBLAS

La biblioteca *Basic Linear Algebra Subprograms* (BLAS) de CUDA, també anomenada cuBLAS, atorga als programadors una gran varietat de funcions d'operacions algebraiques, on es fan operacions costoses de la forma més eficient possible. Per exemple, la multiplicació de dues matrius pot ser una operació costosa, sobretot en matrius amb una dimensió elevada. Amb la funció `cublas_Sgemm()` es pot fer l'operació de forma òptima i el programador només ha de passar-li les matrius i la dimensió. Per a utilitzar la biblioteca cal incloure el fitxer `cublas_v2.h` i quan es compile s'ha de posar el *flag* `-lcublas` (veure [30], de la pàgina 190 – 192). Per veure totes les funcions disponibles es pot consultar [17].

cuRAND

CUDA Random Number Generation library (cuRAND) és una biblioteca que s'encarrega de generar un número de coma flotant de simple o doble precisió de manera aleatòria a través de la GPU. Aquesta acció, que pot ser una mica costosa en la CPU, s'aconsegueix fer en uns pocs cicles en la GPU. Per a utilitzar totes les funcions de la biblioteca cal incloure el fitxer `curand.h` al nostre programa i al compilar cal afegir el *flag* `-lcurand`. (veure [30], de la pàgina 201 – 207). Per veure totes les funcions que es poden utilitzar amb aquesta biblioteca es pot consultar [18].

2.5 Eines

En aquesta secció es detallen les diferents eines que s'han utilitzat al llarg del projecte per poder identificar possibles errors d'implementació o eficiència.

NVIDIA Visual Profiler

Aquesta ordre permet veure per pantalla el nombre de crides i el temps d'execució de cada funció en la GPU. A més, hi ha una aplicació que permet visualitzar d'una forma més interactiva l'execució del programa, ja que compta amb una línia temporal on mostra totes les funcions executades, un resum amb el temps total del programa, el percentatge d'ocupació de la GPU, etc. Per últim, també permet fer una anàlisi de l'execució, amb la qual es pot veure l'ús de memòria, la latència, etc. Per veure més informació es pot consultar [22].

Un exemple d'execució de l'ordre és:

```
$nvprof ./programa
```

O bé, també podem llançar aquest per a després visualitzar les dades en l'aplicació. Per aconseguir-ho s'ha d'exportar el *profiler* a un fitxer amb extensió nvvp.

```
$nvprof -export-profile FILE.nvvp ./programa
```

cuda-memcheck

L'ordre `cuda-memcheck` és una ferramenta de detecció d'errors en temps d'execució per a aplicacions CUDA. Aquesta pot detectar els accessos fora de memòria, desbordaments,

erros d'assignació, etc. En el projecte s'ha utilitzat per trobar possibles errors d'accessos a memòria, ja que no són visibles a simple vista. L'ordre per a executar-la és:

```
$cuda-memcheck [opcions] ./programa
```

Per veure més informació sobre aquesta es pot consultar [19].

tegrastats

S'empra per comprovar els recursos que s'estan utilitzant en el dispositiu NVIDIA Jetson AGX Xavier de manera periòdica, com la quantitat de nuclis CPU o de RAM, el percentatge d'ús de la CPU i la GPU, el consum d'energia que s'està consumint, etc. En el projecte s'ha utilitzat per saber el que consumeix un programa i intentar optimitzar-lo i que siga menys costós o per veure quin mode de potencia s'adapta millor. A banda, l'ordre compta amb diferents opcions, com mostrar la informació a cada determinat temps o que guarde l'eixida en un fitxer, entre altres. A continuació es mostra un exemple d'execució on es posa l'opció per a que mostre la informació per pantalla a cada 10.000 mil·lisegons. Per veure més informació es pot consultar [23].

```
$tegrastats --interval 10000

RAM 2080/15819MB (lfb 3135x4MB) SWAP 0/7910MB (cached 0MB) CPU [46%@1190,
31%@1190,off,off,off,off,off,off] EMC_FREQ 0% GR3D_FREQ 0% AO@41C GPU@42C
Tdiode@44.75C PMIC@100C AUX@41C CPU@42.5C thermal@41.75C Tboard@43C GPU
617/617 CPU 771/771 SOC 1234/1234 CV 0/0 VDDRQ 0/0 SYS5V 1802/1802
```

GitHub

GitHub[24] és una plataforma que permet compartir el codi font d'un programari amb qualsevol usuari d'internet. Els repertoris, on s'emmagatzema el codi, poden ser públics o privats. En cas de ser privats tan sols podran veure el codi el propietari i aquells que tinguen permís. A més, es permet col·laborar amb altres persones per mitjà de la creació de branques dins del repertori a fi de que cadascu publique el seu treball sense modificar els dels altres.

El codi d'HELENNA està publicat en aquesta plataforma, però de manera privada, perquè tan sols puguen accedir els col·laboradors de l'aplicació. El repertori està dividit en branques on cada membre puja i actualitza els seus fitxers. En el nostre cas es creà una branca per pujar tots els fitxers que formen el mòdul cublas16 i, així, tant els tutors com els altres membres del GAP pogueren visualitzar tot el treball realitzat.

Com que GitHub permet publicar el codi en la seua plataforma des de la terminal de Linux, les ordres que s'han fet servir són:

- `git status`: per veure la informació de la branca.
- `git add`: per seleccionar els fitxers que es volen posar o actualitzar en la branca.
- `git commit`: per confirmar que es vol pujar els fitxers seleccionats i permet afegir un missatge.
- `git push`: publicar els fitxers en la branca i siguen visibles per als altres usuaris.
- `git pull`: per descarregar els últims canvis que s'han produït en la branca.

[24] https://github.com

CAPÍTOL 3
Contribució a HELENNA

A continuació, s'explica tot el treball realitzat en HELENNA, es detallarà com s'ha preparat l'entorn per a crear un nou mòdul que utilitze els nombres reals de mitja precisió i la implementació de les diferents capes que formen l'aplicació. En aquest cas la capa completament connectada, la convolucional i la de normalització. També es comentaran els obstacles que s'han trobat al llarg d'aquest treball i com s'han solucionat.

3.1 Preparació prèvia de l'entorn

En aquesta secció s'explica com es va afegir el nou mòdul a l'aplicació i tots els fitxers per poder fer que funcionara. En HELENNA ja existeix un mòdul que utilitza la biblioteca cuBLAS i les operacions es fan amb nombres reals de simple precisió. A més, aquesta versió implementa totes les capes que es descriuen més endavant, és per això que per a crear el nou mòdul i totes les funcions es va seguir aquesta implementació com a guia.

En primer lloc seguint la guia d'HELENNA, on s'indica com crear un nou mòdul, s'afegí en el fitxer `globals.h`[1] una variable de tipus *extern int* anomenada *use_cublas16* per a l'hora de crear les funcions indicar que s'està utilitzant el nou mòdul, el qual s'anomena cublas16. D'altra banda, també es va definir la macro «`#define DEV_CUBLAS16`», per indicar el nom del mòdul.

Una vegada afegides al fitxer `main.c` es va definir una variable `int` anomenada *use_cublas16* per a indicar que la funció a executar ha de ser la implementada en el nou mòdul si aquest està activat. Després, cal posar dues crides, una per a inicialitzar el mòdul i una altra per a indicar que l'aplicació ha acabat amb ell. Per acabar, s'ha d'afegir l'opció `-cublas16` en la funció que imprimeix totes les opcions que hi ha en HELENNA i en `fn_progress_arguments()` per a que quan es llance l'aplicació es puga executar les funcions del nou mòdul.

En segon lloc, es creà els fitxers propis on estaran totes les funcions que s'executaran amb cublas16, en el nostre cas, es crearen quatre fitxers:

- `cublas16.cpp`: aquest fitxer contindrà totes les funcions amb les operacions necessàries per a poder executar les tres capes principals d'HELENNA. El fitxer empra l'extensió cpp, perquè el llenguatge C no accepta el mateix tipus de declaració que CUDA per a referir-se als nombres de mitja precisió. En CUDA s'utilitza *__half* per definir una variable de nombres reals de mitja precisió, mentre que en C es deu definir com *__fp16*. Aleshores s'utilitza l'extensió del llenguatge C++, perquè aquest

[1] Els fitxers amb extensió h són arxius de capçalera. Aquest conté les declaracions directes de les classes, subrutines, variables o altres indicadors.

sí que accepta la mateixa definició que CUDA. Així evitem problemes d'incompatibilitat entre els diferents llenguatges que formen HELENNA.

- `cublas16.hpp`: en aquest es posaran totes les definicions de les funcions del fitxer `cublas16.cpp` per a que arxius externs puguen cridar a aquestes funcions. El fitxer té l'extensió hpp perquè s'utilitzen els fitxers del llenguatge C++.

- `cublas16_kernel.cu`: aquest fitxer contindrà tot el codi que executarà la GPU, és a dir, els diferents *kernel* i les funcions llançadores d'aquests. Per a aquest arxiu s'utilitza l'extensió cu, perquè les funcions *kernel* i les seues crides han d'estar definides dins d'arxius de tipus CUDA.

- `cublas16_kernel.h`: ací es posaran totes les definicions de les funcions del fitxer `cublas16_kernel.cu` perquè fitxers externs puguen cridar a les funcions llançadores. Aquestes crides es faran, majoritàriament, des del fitxer `cublas16.cpp`.

Una vegada creats, es van agafar els quatre fitxers del mòdul de cublas per copiar les funcions al nostre, per a tindre com a referència totes les funcions necessàries. Una vegada copiades s'esborrà el cos de cada una i s'afegí un `printf()` que mostra per pantalla que la funció no està implementada i un `exit(1)`, perquè quan s'execute la funció acabe l'execució. Un exemple, seria el següent:

```
EXTERN_C int fn_vect_mult_cublas16(void *a, void *b, void *c, int n)
{
    printf("fn_vect_mult not implemented yet\n");
    exit(1);
}
```

Per últim, es modificà el fitxer `CMakeList.txt` on s'indica quins dispositius, els fitxers i biblioteques s'han d'utilitzar perquè el compilador els compile i es puguen llançar a execució. Per tant, basant-se en cublas, s'afegiren els quatre fitxers descrits anteriorment i l'opció per indicar que es vol utilitzar cublas16, així es compilen els fitxers que el formen. També es va afegir l'opció per a que quan s'active aquest mòdul es realitze una recerca pel sistema de tots els fitxers de CUDA i de la biblioteca cuBLAS, per veure que estan totes les eines instal·lades en el sistema. A més, s'afegeix el *flag* `-arch=sm_70` ([6], pàgina 90) per indicar l'arquitectura de la GPU i així poder fer les operacions amb els nombres reals de mitja precisió. Si aquest *flag* no es posa, la GPU no reconeix aquest tipus de codi i no l'executa.

3.2 Capa completament connectada

La capa completament connectada o *Fully Connected* és un tipus d'estructura de xarxes neuronals on cada neurona està connectada a cada neurona de la següent capa [15], un exemple d'aquesta seria la figura vista en la introducció (Figura 1.2). A ser la primera a implementar-se en el mòdul cublas16, es va haver d'implementar totes les funcions inicials com la d'inicialització, en la qual es generen els *streams*[2]; la de finalització on es destrueixen els *streams* creats; o la funció d'error que et mostra quin error s'ha produït en la GPU. Per altra banda, també s'importaren tots els fitxers externs que farien falta per al desenvolupament del treball, com la referència a la biblioteca cuBLAS, als distints fitxers

[2]Els *streams* són com una cola d'operacions de la GPU que s'executen amb un ordre específic. Aquestes operacions poden ser llançaments de *kernel*, traspassos de memòria, etc. ([28], pàgina 192).

de l'aplicació, al fitxer que permet utilitzar les operacions aritmètiques definides per a *__half*, entre altres.

Una vegada el fitxer tenia totes les funcions inicials implementades, se seguí una metodologia de llançar l'aplicació amb una determinada xarxa neuronal, en aquest cas s'utilitzà la *mlp_tiny*, la qual utilitza la base de dades MNIST i tan sols està composta per una capa completament connectada amb una mida xicoteta, perfecta per identificar les funcions que han d'executar-se i trobar fàcilment els errors d'implementació. Aleshores, es llançava a execució l'aplicació amb aquesta xarxa neuronal i com s'havien posat els *printf()* i els *exit(1)*, quan parava l'execució s'implementava la funció en la qual havia parat. Així, fins que l'aplicació s'executava completament. També, recalcar que com el tipus *__half* no està suportat pel llenguatge C i HELENNA està programat majoritariament amb aquest llenguatge, es va haver d'utilitzar el tipus de dada *void*[3] per a que no hi haguera problemes d'incompatibilitat entre les funcions. També es canviaren el tipus de variables que es passaven en les crides a les funcions del fitxer `cublas16.cpp` per a que foren de tipus *void*.

Les primeres funcions a implementar van ser les d'assignació, escriptura, lectura i alliberament de memòria. En la funció d'assignació es passa com a paràmetre la grandària del *buffer* i aquest està calculat de manera general per a totes les funcions de l'aplicació. Aleshores, el tipus de dada que s'utilitza per a calcular la mida és el *float*, que ocupa 4 bytes, com s'ha mencionat en el capítol anterior. Açò era un problema, ja que s'havien d'assignar *buffer* que tindrien valors de tipus *__half* i aquests ocupen 2 bytes de memòria. Per tant, la meitat del *buffer* es quedaria buit i acabaria sent un desaprofitament de memòria. Com a solució, es dividia per la meitat la mida passada a la funció, així no es quedaria mig *buffer* buit. Pel que respecta a la funció d'escriptura i lectura, HELENNA treballa completament amb nombres de simple precisió, aleshores, s'havia de convertir les dades a l'hora d'assignar-les en la GPU. Per realitzar aquestes conversions es feia amb la crida:

- `__float2half()`: per convertir un valor *float* a *__half*.
- `__half2float()`: per convertir un valor *__half* a *float*.

Aquestes ordres s'apliquen sobre un *buffer* auxiliar per a després passar les dades a la GPU a través de `cudaMemcpy()`.

Una vegada implementades totes les funcions anteriors, es va adaptar el fitxer `mem_allocate.c` perquè HELENNA cridara a les funcions i s'emmagatzemaren identificadors dels *buffers* en l'estructura que té definida. Per crear-la s'afegí a les funcions d'escriptura, lectura, assignació i alliberament la crida a les respectives funcions del fitxer `cublas16.cpp`.

Quan totes les funcions de gestió de memòria estaven implementades es procedí a implementar les funcions que feien operacions matemàtiques, com bé s'ha dit abans, aquestes es guiaven per la implementació realitzada en la versió de cublas i, com en moltes ocasions, si no es podia realitzar una implementació similar, es gastava com a guia el fitxer `cpu.c`. Aquestes situacions ocorrien quan en la versió de cublas s'utilitzaven funcions de la biblioteca cuBLAS, les quals estan optimitzades per a realitzar la funció desitjada. Aquestes funcions abasten una gran quantitat d'operacions algebraiques, però en la majoria d'elles no contemplen els nombres de mitja precisió. Per tant, quan les funcions estaven implementades per funcions de la biblioteca cuBLAS, es devia fer de manera «manual» per aconseguir fer l'operació desitjada. En aquests casos, s'accedia a

[3] La dada de tipus void, no representa cap valor, és com una variable buida, la qual no es pot utilitzar per a fer operacions.

la documentació de la biblioteca [17] per veure que feia la funció que utilitzava la versió cublas i s'implementava a través de les funcions matemàtiques per a la dada *__half*, desenvolupat per NVIDIA [21]. En altres ocasions també es recorria a l'arxiu «cpu.c» per veure la implementació i programar una nova versió amb els nombres de mitja precisió. Uns exemples d'aquestes dues situacions descrites són:

- Funció matmul: en aquesta funció es realitza una multiplicació entre dues matrius i està implementada amb l'única funció de la biblioteca cuBLAS que suporta els nombres de mitja precisió, la funció `cublasgemm()` (veure [17], pàgina 70). Aquesta el que fa és multiplicar una fila de la matriu A per una columna de la matriu B, el resultat, el suma a la matriu C. La qual serà la matriu que emmagatzeme el resultat (veure Figura 3.1). La implementació d'aquesta funció es pot consultar en l'apèndix A.

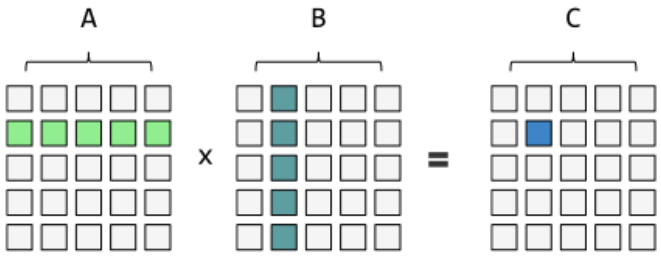

Figura 3.1: Representació de l'operació que realitza la funció matmul. En aquest cas es tenen dues matrius quadràtiques on es multiplica element a element la fila de la matriu A per la columna de la matriu B i després se sumen tots aquests valors, els quals són guardats en la matriu C.

- Funció matadd: la funció implementa l'operació axpy, és a dir, es fa una multiplicació d'un vector (a) amb un escalar (x) i el resultat de la multiplicació se suma a un altre vector (y). En aquesta funció es volia utilitzar la funció `cublasgeam()` (veure [17], pàgina 98) que implementa l'operació axpy, però aquesta funció no és compatible amb el tipus de dada *__half*. Aleshores, es va veure en què consistia l'operació i es realitzà a través d'un *kernel*. Per a implementar-la s'utilitzaren les crides `__hmul()` per a la multiplicació de *half* i `__hadd()` per a la suma (veure [21], pàgines 3 i 4). Per veure el codi es pot consultar l'apèndix A.

Una vegada vist un exemple de cada escenari descrit, en les Taules 3.1 i 3.2 es llisten totes les funcions implementades en aquesta capa i una xicoteta descripció de cada una.

Quan s'implementaren totes les funcions que componien la capa, es va procedir a comprovar que no hi havia cap error d'assignació en memòria, perquè quan es llançava l'aplicació a execució no superava el 10% de precisió i no es trobava la causa exacta que podia originar aquest comportament. Aleshores, s'utilitzà l'eina `cuda-memcheck` i es descobrí que hi havia *buffers* que s'assignaven de manera incorrecta en memòria. Per tant, es va haver de solucionar, i una vegada resolt l'error, la precisió de l'aplicació era similar al obtingut pel mòdul cublas.

3.3 Capa convolucional

Una vegada implementada la capa completament connectada i veure que donava resultats millors que els presentats pel mòdul cublas, es procedí a implementar la capa convolucional. Aquesta capa s'enfoca en l'àrea de la visió per computador, és a dir, se centra en

Funció	Descripció
`allocate_buffer`	Es realitza l'assignació dinàmica de memòria en la GPU.
`deallocate_buffer`	S'allibera la regió de memòria de la GPU associada a un punter.
`read_buffer`	Copia les dades des de la GPU a la CPU i els transforma de *__half* a *float*.
`write_buffer`	Copia les dades des de la CPU a la GPU i els transforma de *float* a *__half*.
`write_value`	S'emmagatzema en la posició indicada del vector el valor passat com a paràmetre.
`init_params`	S'inicialitzen els paràmetres, en aquest cas es els *streams*.
`zero_vec`	Es guarda el valor 0 en cadascuna de les posicions d'un vector.
`set_vec`	S'emmagatzema en totes les posicions del vector el valor passat com a paràmetre.
`mat_vec_mult_row`	Es multiplica un vector per cada fila d'una matriu.
`mat_set`	S'assigna un valor a tota la matriu passada per paràmetre.
`vec_axpy`	Es fa l'operació axpy en un vector.
`matmul`	Multiplicació de dues matrius.
`matmul_bt`	Multiplicació de dues matrius, on la segona està transposada.

Taula 3.1: Funcions implementades en la capa *fully connected*, part 1.

el processament d'imatges i permet classificar-les o identificar els objectes que hi ha en elles. Per realitzar aquestes accions es combina la capa completament connectada, la convolucional i la maxpooling. Com que la primera ja estava implementada es programaren les funcions necessàries per a executar una xarxa neuronal d'aquest tipus. Cal recalcar que les següents funcions, que seran explicades a continuació, estan implementades en el mòdul cublas utilitzant *kernels*. Aleshores, en el nostre cas, s'ha fet el mateix i les funcions no presenten un gran canvi entre mòduls.

En primer lloc, es procedí a la implementació de la funció `maxpooling`. Aquesta funció fa l'operació *maxpooling* ([7], pàgina 190), és a dir, es realitza un filtre sobre una imatge passada com a paràmetre. Aquesta imatge en realitat és una matriu de tres dimensions. Per realitzar el filtre s'agafa un bloc de la matriu i s'escull el valor més gran del bloc per assignar-lo en una matriu d'eixida (veure Figura 3.2). Al final del procés obtenim una matriu molt més menuda que representa l'original, així el tractament de les imatges és menys costos. En el mòdul cublas aquesta implementació consisteix a aplicar comparació i assignació de valors. Aleshores, en aquest cas, es procedí a utilitzar la mateixa implementació que hi havia en el mòdul cublas, però adaptant les operacions per a que es poguera fer amb els valors *__half*.

En la implementació d'aquesta funció es va trobar un inconvenient en els *buffers* que utilitzàvem com a paràmetres d'entrada. En aquest cas un dels *buffers* emmagatzemava l'índex del valor més gran del bloc a filtrar. Aquest fet obligava a fer que els valors foren de tipus enter, ja que per accedir a les posicions determinades d'un vector és necessari un enter per a indicar la posició. Aleshores, s'havia de modificar la funció d'assignació, escriptura i lectura, per a que quan no fóra necessari realitzar una conversió o assignar *buffers* que contingueren dades de tipus *__half*, es poguera realitzar. Dit d'una altra manera, calia crear unes accions específiques per a poder tenir altres tipus de *buffers*. Com a solució es va afegir un paràmetre d'entrada a les funcions, anomenat `keep_size`. Si

Funció	Descripció
`matmul_at`	Multiplicació de dues matrius, on la primera està transposada.
`matmul_elwise`	Es realitza la multiplicació de dues matrius, però com si foren vectors.
`vect_scalar_product`	Multiplicació d'un nombre escalar per un vector.
`vect_mult`	Multiplicació de dos vectors.
`matadd_col`	Donada una matriu, se suma tots els elements en cada columna i el valor s'emmagatzema en un altre vector.
`matadd`	Suma de dues matrius, a través de l'operació axpy.
`matsub`	Resta de dues matrius, a través de l'operació axpy.
`mat_reduce_rows`	Es rep una matriu com entrada i se suma tots els elements de cada fila i es guarda en un vector d'eixida.
`matrix_relu`	Funció d'activació ReLU.
`matrix_relu_der`	Funció d'activació ReLU, però torna *True* si el valor és positiu i *False* si el valor és negatiu.
`matrix_softmax`	Funció d'activació softmax.
`vec_copy`	Còpia de tots els elements d'un vector a un altre.
`copy`	Còpia de tots els elements d'una matriu a una altra.
`check_status`	Funció per imprimir els errors que poden tornar les funcions de la biblioteca cuBLAS.
`check_malloc_status`	Funció que imprimeix els errors ocasionats per l'assignació de memòria en GPU.
`synchronize`	La crida a aquesta funció realitza la sincronització entre CPU i GPU.

Taula 3.2: Funcions implementades en la capa *fully connected*, part 2.

aquest valor era un 1, el *buffer* havia de ser assignat amb la mateixa mida amb la què es rebia, en cas de l'escriptura i lectura, no es devien transformar els valors a l'altre tipus de dada. En el cas que el valor fora 0, s'havia de fer les conversions i assignar en memòria la mida corresponent a *__half*.

Una vegada programada la funció i solucionat el problema, es continuà amb la implementació de la capa. En aquest cas es va fer la funció `demaxpooling` on es realitzava la funció contrària a la funció `maxpooling`. Per tant, la matriu que representava la imatge es deixava com estava abans de fer l'operació *maxpooling*. Després, s'implementà la funció `im2col`, la qual converteix les imatges en columnes. Per ser més específics s'agafa la matriu de tres dimensions, que representa la imatge, i la converteix en diferents columnes, formant una única matriu de dues dimensions (veure Figura 3.3). A continuació, es programà la funció `col2im` on es feia la mateixa operació que la funció `im2col`, però a la inversa.

Per últim, es llançà a execució una xarxa neuronal convolucional i, comparant-lo amb els resultats del mòdul cublas, s'observà que la implementació realitzada era més costosa i menys eficient. Aleshores, s'utilitzà l'eina `nvprof` per veure tota l'execució de l'aplicació i intentar trobar la raó per la qual era ineficient. Açò ajudà a identificar les funcions que tardaven més del que devien i poder identificar on estava el possible problema. Es va arribar a la conclusió que el problema es podia originar perquè les variables no estarien en memòria conjunta i que els accessos a memòria per obtenir les dades foren més lentes que en la versió de cublas.

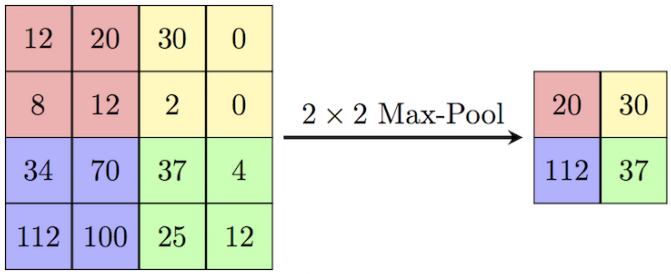

Figura 3.2: Exemple visual de l'operació que realitza la funció `maxpooling`, en aquest cas s'agafen blocs de quatre elements i s'escull el valor més elevat. A banda, també considerar que la matriu resultant d'aquesta operació serà del mateix mida que els blocs que s'agafen, en aquest cas de 2x2.

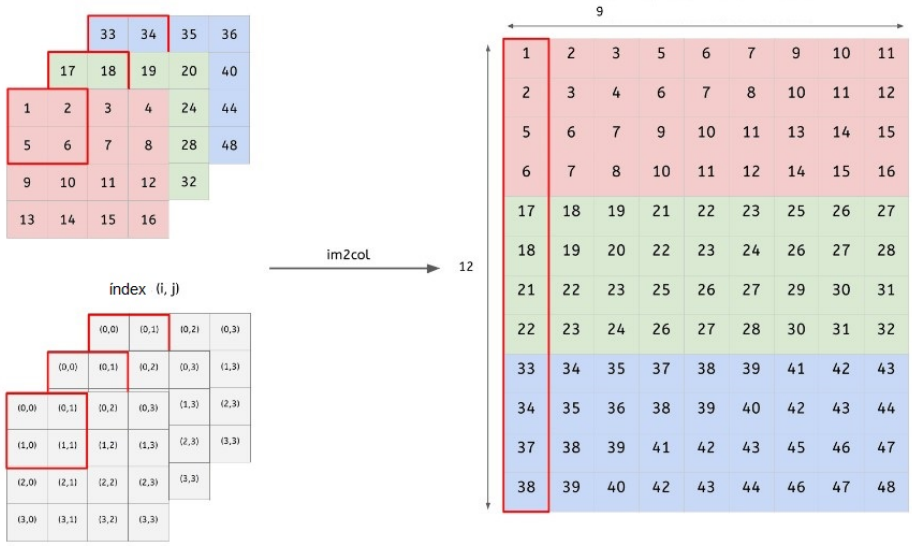

Figura 3.3: Exemple visual de l'operació im2col que realitza la funció `im2col`, on en aquest cas es veu com una matriu 3D es transforma en una matriu 2D. Per a fer-ho s'agafen blocs de 4 elements i es posen en forma de columna. Així successivament fins que tots els valors formen una nova matriu de dos dimensions.

3.4 Capa de normalització

Aquesta capa té l'objectiu de normalitzar les eixides de les neurones de les capes ocultes per a minimitzar l'error. En HELENNA hi ha dos tipus, la *Batch Normalization* i la *Dropout*.

Batch Normalization

Dels diferents tipus de capes de normalització, la capa *Batch Normalization* [31] és la que millors resultats i beneficis aporta a l'entrenament de xarxes neuronals denses, és a dir, xarxes neuronals amb una gran quantitat de capes. Aquesta proporciona a l'execució d'una xarxa neuronal, sobretot en les convolucionals, una millor precisió i permet un entrenament en menys temps. Aquesta capa consta de les funcions que es poden veure en la Taula 3.3, on es descriu la funcionalitat de cada una.

Quan es van implementar totes les funcions, es va llançar a execució una xarxa neuronal que utilitzava aquesta capa per comprovar que les funcions implementades funcionaven correctament. Com a resultat s'obtenia una precisió de menys del 10%, aleshores s'activà una funció que hi ha a HELENNA per veure els valors dels *buffers* mentre s'exe-

Funció	Descripció
`batch_normalization`	Funció que calcula la mitjana i la desviació estàndard del lot actual i s'aplica a l'eixida per a normalitzar-la.
`batch_normalization _inference`	Funció similar a l'anterior, però s'utilitza una mitjana mòbil i la desviació estàndard és la que s'ha vist al llarg de tot l'entrenament.
`batch_normalization _compute_gradients`	Es calculen els gradients, gamma i beta, on gamma és un factor d'escala i beta un factor de desplaçament.
`batch_normalization _backward`	En aquesta funció es fan les mateixes operacions que en la `batch_normalization`, però en sentit contrari. Així es reverteixen les operacions realitzades.

Taula 3.3: Funcions implementades en la capa *Batch Normalization*.

cuta l'aplicació. Seguint la traça de l'execució s'observà que en un determinat moment les operacions operaven amb valors NaN i infinits. Com a conseqüència, se seguí la traça de l'execució per veure en quina funció exactament s'originaven aquests valors i s'arribà als *buffers* que s'utilitzen en la funció `batch_normalization`. Com que dins d'aquestes funcions es fan diverses crides a altres, es posaren `printf()` perquè s'imprimiren per pantalla els elements dels *buffers* que entraven a les distintes funcions i poder aïllar el problema. Al final, es trobà una funció que podia originar els infinits i NaN. Aquesta es traslladà a un entorn controlat per veure si feien les operacions de forma correcta i s'arribà a la conclusió què la seua implementació sí era correcta. Així, es procedí a fer manualment l'operació amb les dades dels *buffers* d'HELENNA, es descobrí que per a calcular la variància es realitzaven diverses funcions intermèdies i en aquestes s'obtenien valors que sobreïxen del rang de representació del *__half* i per tant el valor es guardava com un $+\infty$ o $-\infty$.

Quan es descobrí el problema real de la baixa precisió es va intentar canviar la implementació per evitar les sobreeixides del rang. Açò no va ser possible i com a solució es proposà fer aquesta capa de forma mixta, és a dir, les operacions realitzades en totes les funcions d'aquesta capa es farien en *float*. Aleshores quan es fa una crida a aquestes funcions s'hauria de convertir els valors a nombres de coma flotant de simple precisió.

Dropout

Per últim, s'implementà la capa *Dropout* [29], la qual suavitza les eixides de les neurones ocultes. Aquesta acció la fa desactivant neurones, tant de les capes visibles com de les ocultes, i les seues respectives entrades i eixides. Aquesta acció és de forma temporal i l'elecció de quines s'han de desactivar es fa de manera aleatòria. Per a decidir quina neurona es desactiva o no es crea un *buffer* amb nombres aleatoris i si el nombre és menor que el llindar establit, la neurona no es desactiva. Però si és major, aquesta es desactivarà.

En HELENNA la funció que realitza l'acció de desactivar les neurones és l'anomenada com `matset_random_ones` on primer es creen els valors aleatoris en un *buffer* auxiliar. Després, es fa la comparació per determinar si la neurona es desactivarà o no. Per ser més precisos, aquesta funció es divideix en dues parts, una és executada en la CPU i l'altra en la GPU. La generació de nombres aleatoris es realitzà a través de la CPU i la comparació d'aquests nombres amb el llindar s'executa en la GPU. Cal recalcar que aquesta funció necessitava ser optimitzada, ja que l'execució d'aquesta funció en el mòdul cublas16 tenia un cost temporal més elevat comparat amb el mòdul cublas. Aquesta optimització es detallarà en la següent secció.

Versió	Temps d'execució	Temps de conversió
CPU	11.41	4.18
CPU amb OpenMP	10.81	1.8
GPU	15.07	3.36

Taula 3.4: Execucions realitzades amb la xarxa neuronal mlp_tiny que utilitza la base de dades MNIST. Es presenten el temps d'execució total del entrenament i el temps que tarda a fer-se la conversió. Aquestes dades es presenten en segons.

3.5 Optimització de funcions

Quan totes les funcions de totes les capes estaven implementades i totes entrenaven i donaven una precisió similar al mòdul de cublas, es procedí a comparar les execucions dels dos mòduls, funció a funció. Graciés a HELENNA, aquest treball fou més senzill, ja que en finalitzar l'execució mostra per pantalla el nombre total de crides, el temps total i el temps per crida. L'objectiu d'açò era trobar totes les funcions del mòdul cublas16 que foren més costoses que les de cublas. Amb aquesta cerca es trobaren diferents funcions que tenien un alt cost temporal, aleshores, es procedí a optimitzar-les. No s'ha pogut acomplir aquesta tasca en totes les funcions que necessiten ser optimitzades per qüestions de temps. A continuació s'expliquen totes les funcions que si ha sigut possible optimitzar.

Operació de conversió

En primer lloc, s'observà que el mòdul cublas no tenia cap funció de conversió de valors, ja que no era necessari convertir els valors a uns altres, però com en la nostra versió si era necessari, es va haver d'implementar. L'execució de les conversions es feia a través de la CPU, és a dir, el codi s'executava en la CPU. Com açò era un cost temporal afegit en comparació amb l'altre mòdul, es va voler optimitzar-lo per a reduir el seu temps d'execució al mínim. Primer s'intentà fer les conversions a través de la GPU. Aquesta versió no va resultar del tot òptima, ja que era més costosa que la versió de CPU. Aleshores, es decidí optimitzar el codi a través d'OpenMP.

Com es pot veure en la Taula 3.4, el temps d'execució total de la funció de conversió era de 4.28 segons en un principi. Amb la implementació del *kernel* el temps per funció es disminuí un poc, però el temps d'execució total augmenta quasi 4 segons. La qual cosa indica que aquesta implementació era pitjor, ja que a l'hora de cridar a les funcions de la GPU, aquesta ha de preparar els valors i assignar-los als fils corresponents. Açò suposava un cost temporal, el qual feia que empitjorara de manera global. Amb OpenMP s'aconsegueix un menor temps en cada funció, però aquesta millora no és tan significativa en el temps total. Igualment, és la millor versió que es pot aconseguir i en xarxes neuronals més grans pot implicar una millora significativa. El codi resultant es pot consultar a l'Apèndix B.

Funció `matadd_col`

Després, s'observà que la funció `matadd_col` (veure Taula 3.2) suposava un alt cost temporal en xarxes neuronals grans. Com que aquesta funció estava implementada a través d'una crida *kernel*, es va haver de donar un altre enfocament per a aconseguir un millor temps d'execució. Per saber com canviar-ho i programar-ho perquè s'utilitzaren els fils de la millor forma possible, es consultà [20], un arxiu de NVIDIA on s'explica com realitzar un codi eficient.

Versió	Temps d'execució	Temps de cada crida
Mòdul cublas	221.23	55.04
Mòdul cublas16 sense optimització	177.61	101.83
Mòdul cublas16 amb optimització 1	185.39	217.22
Mòdul cublas16 amb optimització 2	174.74	48.33

Taula 3.5: Execucions realitzades amb la xarxa neuronal mlp_large que utilitza la base de dades MNIST. Es presenten el temps d'execució total, en segons, i el temps que tarda a fer-se cada crida a la funció, en microsegons. Per altra banda, «Mòdul cublas16 amb optimització 2» té una assignació de blocs on hi ha el mateix nombre de blocs que files tinga la matriu d'entrada. El «Mòdul cublas16 amb optimització 2» té una assignació on el nombre de blocs són el resultat de dividir el nombre d'elements pel nombre màxim de fils per bloc.

Versió	Temps d'execució	Temps de cada crida
Mòdul cublas	130.68	1.21
Mòdul cublas16 en CPU	199.79	48.80
Mòdul cublas16 amb cuRAND	147.29	4.83

Taula 3.6: Execucions realitzades amb la xarxa neuronal vgg3-Dropout que utilitza la base de dades Cifar-10. Es presenten el temps d'execució total i el temps que tarda a fer-se cada crida de la funció. Aquestes dades es presenten en segons.

En la Taula 3.5, es poden veure les diferents execucions realitzades i el temps que tardava cada una. A més, es veu la importància del nombre de blocs que s'assignen a un determinat problema, ja que depenent del nombre de blocs que es tinga, el temps d'execució pot ser major o menor. Per ser més exactes, si es veu la versió «Mòdul cublas16 amb optimització 1» el temps que tarda cada crida és de 217.22 microsegons, mentre que el temps de la versió «Mòdul cublas16 amb optimització 2» és de 48.33, el que suposa un 78% menys. Si ho comparem amb la versió cublas, és una mica menys costosa, però respecte a la versió antiga de cublas16 tardà la meitat de temps. El codi es pot consultar a l'Apèndix B.

Funció `matset_random_ones`

Quan es va implementar la capa de normalització *Dropout* s'executà per veure el temps d'execució de la xarxa neuronal, quan utilitzava aquesta implementació, i s'observà que el temps era major que amb el mòdul cublas. En conseqüència, es buscà una alternativa per poder reduir el cost temporal. Com que els nombres aleatoris es feien en CPU es decidí passar aquesta tasca a la GPU i veure si s'aconseguia una millora. Per a crear els nombres aleatoris s'utilitzà la biblioteca cuRAND. Però, aquesta biblioteca no té funcions en les quals es puga generar valors amb el tipus de dada *__half*, aleshores, es va haver d'utilitzar un *buffer* auxiliar on generar els nombres aleatoris en format *float* i després convertir-los a *__half*. En la Taula 3.6 es pot veure com la versió que fa l'operació en CPU tarda 48.8 segons i la versió que utilitza cuRAND tarda 4.83, el que suposa una millora de 10×. També cal recalcar que aquesta versió no aconsegueix un millor temps que el mòdul cublas, però es redueix el cost temporal notòriament. El codi es pot consultar a l'Apèndix B.

CAPÍTOL 4
Resultats obtinguts en HELENNA

En aquest capítol es presenten i s'analitzen les diferents gràfiques amb les dades obtingudes dels entrenaments realitzats en HELENNA. En aquestes gràfiques es mostra el temps d'execució, el cost energètic i la precisió obtinguda.

4.1 Entorn d'execució

En aquesta secció es detalla com s'executa els entrenaments amb l'aplicació i els paràmetres més importants que s'han utilitzat per a les ordres d'execució. A més, es detalla com s'ha calculat el cost energètic de cada entrenament.

Per obtenir totes les dades que s'exposen en les següents seccions s'ha executat l'aplicació amb una xarxa neuronal formada per capes completament connectades (mlp_large), una altra amb la capa convolucional i la completament connectada (conv-16channels). Per últim, una xarxa neuronal amb les dues anteriors, més la capa de normalització *Dropout* (vgg3-dropout).

Per llançar aquestes xarxes neuronals en HELENNA es necessiten diferents paràmetres per a configurar l'entrenament. Aquests són:

- Nombre d'èpoques (`-ne`). Indica el nombre de voltes que s'utilitzaran les dades en l'aprenentatge.
- *Learning rate* (`-lr`). Percentatge que determina el canvi de pesos en cada iteració.
- *Minibatch size* (`-mbs`). Indica el nombre de dades que hi haurà per lot.
- El mòdul. En aquest cas, com que s'utilitzen el mòdul cublas i cublas16, es posarà `-cublas` i `-cublas16`.
- Activació de la sincronització entre GPU i CPU (`-force_sync`)

Per exemple, l'ordre que hem fet servir per executar la xarxa mlp_large és:

```
./helenna -cublas16 -net ../nets/mnist/mlp_medium -dataset
  ../datasets/mnist/ -ne 10 -lr 0.1 -mbs 16 -force_sync
```

Amb aquesta ordre s'ha llançat a execució les tres xarxes neuronals, on es canvia el tipus de base de dades i el nom de la xarxa. D'altra banda, aquestes execucions s'han llançat en dos dels quatre modes de potència que presenta l'AGX Xavier. Aquests són la de 15 W i l'EDP (veure Figura 4.1). L'objectiu era presentar les dades en aquests dos

modes de potència, però també amb el mode de 10 W. La qual cosa no ha sigut possible, perquè HELENNA necessita tres nuclis de CPU per poder executar-se i amb aquest mode tan sols se n'activen dos.

Mode ID	0	1	2	3	4	5	6
Power Budget	EDP	10W	15W	30W	30W	30W	30W
Online CPU	8	2	4	8	6	4	2
CPU Max Frequency (MHz)	2265.6	1200	1200	1200	1450	1780	2100
GPU Max Frecuency (MHz)	1377	520	670	900	900	900	900
DLA Max Frequency (MHz)	1395.2	550	750	1050	1050	1050	1050
PVA Cores	2	0	1	1	1	1	1
PVA Max Frequency (MHz)	1088	0	550	760	760	760	760
Max Frequency	2133	1066	1333	1600	1600	1600	1600

Figura 4.1: Amb aquesta taula es pretén il·lustrar els diferents modes de potència de l'AGX Xavier i quins són els elements que s'activen en cada mode. Cal recalcar que el mode de potència EDP és l'únic que no té limitació de consum i és el que permet utilitzar el maquinari al màxim rendiment. Per veure com canviar entre els diferents modes de potència consultar [14].

Per últim, en les seccions següents s'analitzen els resultats obtinguts i es compararan entre els mòduls cublas i cublas16 i el rendiment dels dos modes de potència. Per aconseguir-ho s'utilitzarà el temps d'execució i la precisió que mostra HELENNA per pantalla quan acaba l'execució. A més, es presenten dues gràfiques amb el cost energètic de cada versió. Aquest cost s'ha calculat amb els watts i el temps d'execució de l'entrenament. Per obtenir els watts s'ha utilitzat l'eina `tegrastats`, explicada en la secció 2.5, la qual mostra els mil·liwatts consumits. L'equació que hem fet servir per a obtenir el cost energètic és:

$$\text{cost (J)} = (\text{potència (mW)} \times 10^{-3}) \times \text{Temps d'exe (s)}$$

4.2 Resultats per capes separades

En aquesta secció s'analitzaran les xarxes neuronals formades per capes completament connectades i capes convolucionals. Per a fer aquesta anàlisi s'han obtingut els temps d'execució, el consum energètic i la precisió de cada entrenament.

Xarxa neuronal mlp_large

La xarxa neuronal mlp_large està composta per 21 capes de les quals 10 són capes completament connectades, 9 funcions d'activació ReLU, una capa d'entrada i una altra d'eixida. Aquesta configuració i els paràmetres de cada capa es poden veure a la Figura C.1 de l'apèndix C. També fem notar que aquesta xarxa fa servir la base de dades MNIST.

En la Figura 4.2 es presenta una gràfica amb el temps d'execució mlp_large utilitzant cublas (FP32, en la gràfica) i cublas16 (FP16, en la gràfica). Com s'observa, en qualsevol dels dos modes de potència, la versió de cublas16 presenta un menor temps d'execució que en el mòdul cublas. Per ser més concrets, en el mode de 15 W, cublas16 és un 24.88% més ràpid que cublas i en el mode EDP, cublas16 és un 23.83% més ràpid. Aquesta millora temporal s'ha aconseguit únicament utilitzant els nombres en coma flotant de 16 bits, ja que la implementació de les funcions que les componen són molt similars entre els dos mòduls. Si ens fixem en els modes de potència, es veu clarament una diferència temporal

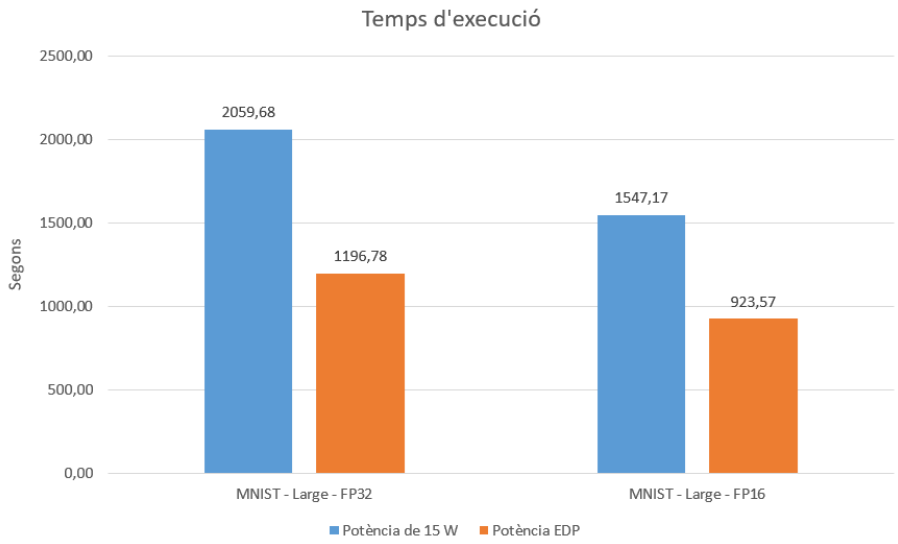

Figura 4.2: Temps d'execució de la xarxa neuronal mlp_large.

per utilitzar més o menys recursos del dispositiu. En el mòdul de cublas, utilitzar el mode de potència EDP implica una acceleració de l'execució d'un 41.89%. En cublas16, ocorre exactament el mateix, amb el mode de potència EDP, s'aconsegueix una acceleració del 40.31% respecte al mode de 15 W.

Després, si ens fixem en la Figura 4.3, es pot veure una gràfica amb el cost energètic de cada execució realitzada. Aleshores, si comparem el consum energètic dins de cada mòdul, en cublas tenim que EDP consumeix un 20.52% menys, i en cublas16 un 23.06% menys. Si comparem entre mòduls, cublas16 presenta una reducció del consum del 31.59% respecte a cublas, quan s'utilitza el mode de potència de 15 W. En el mode EDP, cublas16 consumeix un 33.77% menys.

Per últim, si s'observen les columnes «MNIST - Large - FP32» i «MNIST - Large - FP16» de la Figura 4.6, es mostra la precisió que s'ha obtingut en cada execució. Entre els modes de potència, no hi ha una diferència significativa entre ells, ja que els recursos utilitzats no influeixen en la precisió de l'entrenament. Però, d'aquesta gràfica sí que cal destacar que la diferència entre mòduls no és significativa, perquè tan sols es diferencien per unes dècimes. Açò implica que cublas16 obté quasi la mateixa precisió que cublas i ho aconsegueix amb una major eficiència, tant temporal com energètica. Per tant, per a executar aquesta xarxa neuronal la millor configuració és amb el mòdul cublas16 i el mode EDP. En cas de voler utilitzar el mode de 15 W, el millor mòdul per a realitzar en l'entrenament també és cublas16.

Xarxa neuronal convolucional de 16 canals

La xarxa neuronal convolucional de 16 canals utilitza la base de dades MNIST i està formada per una capa convolucional amb 16 canals, una *maxpooling*, una funció d'activació ReLU, una capa d'entrada, una altra d'eixida i una completament connectada. Per veure la configuració i els paràmetres de cada capa, veure Figura C.2 de l'apèndix C.

En primer lloc, ens centrem en el temps d'execució (Figura 4.4). En aquest cas no es produeixen els mateixos resultats que en la capa completament connectada, ja que per exemple en el mode de 15 W, cublas és un 9.59% més ràpid que cublas16. No obstant això, en el mode EDP s'aconsegueix l'efecte contrari i el mòdul cublas16 és un 20% més ràpid que cublas. Si comparem entre els modes de potència, veiem que amb el mode

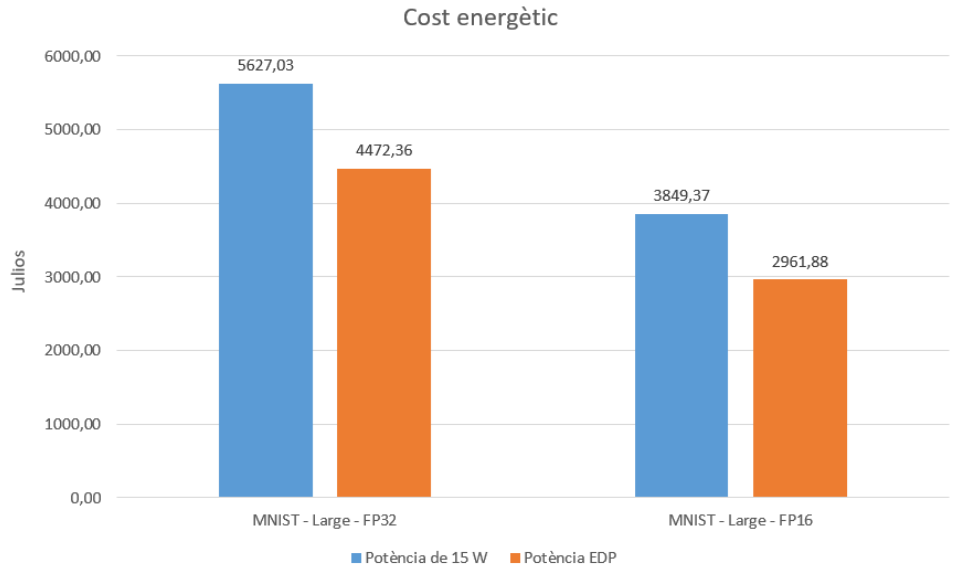

Figura 4.3: Consum energètic de la xarxa neuronal mlp_large.

EDP, cublas aconsegueix un temps d'execució superior (1%) respecte al mode de 15 W, però en cublas16 és un 26.89% més ràpid. Cal recalcar que la majoria d'operacions que es realitzen són de comparació i d'assignació, aleshores, l'increment del temps d'execució es pot deure als temps d'accés a les dades, la qual cosa pot implicar un gran cost temporal.

En segon lloc, la gràfica que representa el cost energètic (Figura 4.5) és molt similar a la gràfica del temps d'execució. En aquest cas, en el mode de potència de 15 W, cublas16 consumeix un 9.59% més, però en EDP, cublas consumeix un 11.29% més. Entre els modes de potència, EDP consumeix un 1.47% més que en 15 W, quan s'utilitza cublas. Però, en cublas16 s'aconsegueix un 17.87% menys de consum.

Per últim, en la Figura 4.6, les columnes «Conv - 16ch - FP32» i «Conv - 16ch - FP16», es pot veure la precisió que s'ha obtingut en les quatre execucions. L'únic remarcable de les quatre és que entre modes de potència no hi ha diferència de precisió i que entre els mòduls, cublas16 obté 0.04% menys de precisió. Tenint en consideració l'alta precisió dels dos mòduls, es pot afirmar que cublas16 fa totes les operacions de manera correcta, però no de forma òptima, ja que, com bé s'ha vist, presenta pitjors dades d'execució i consum energètic, excepte quan s'utilitza el mode EDP que s'aconsegueix l'efecte contrari. És per això que la millor configuració per executar aquesta xarxa neuronal és utilitzar el mòdul cublas16 amb el mode EDP, ja que és el que menys temps i consum energètic presenta. Però si es volguera utilitzar el mode de 15 W, la millor opció seria el mòdul cublas.

4.3 Resultats amb totes les capes en conjunt

En aquesta secció s'utilitzarà una xarxa neuronal que està formada per totes les capes que hem vist en el capítol 3. Com que *Batch Normalization* no té un correcte funcionament la xarxa neuronal que hem utilitzat compta amb la capa *Dropout*. La xarxa neuronal seleccionada per mostrar els resultats és la vgg3-dropout, la qual utilitza la base de dades Cifar-10. Aquesta xarxa és la més gran de les tres presentades i està formada per un total de 24 capes. Aquestes capes són:

- 1 capa d'entrada.
- 1 capa d'eixida.

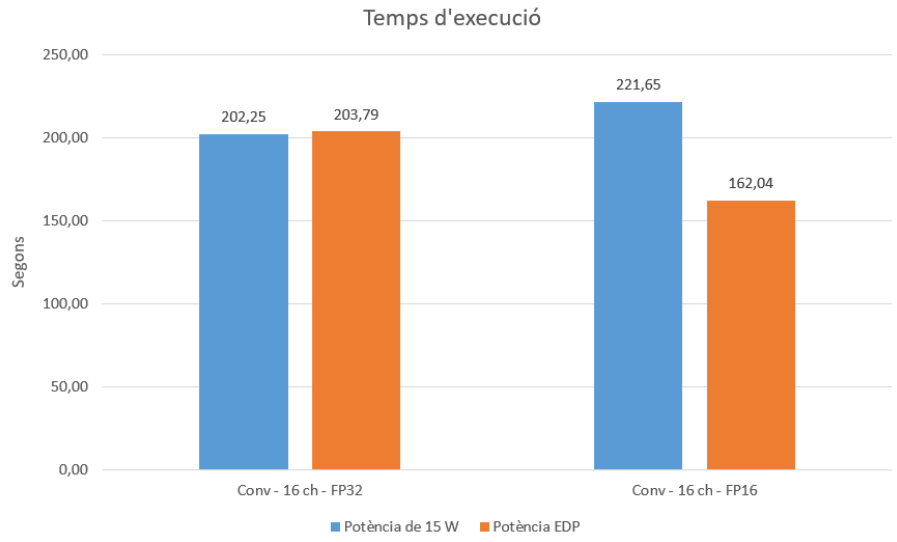

Figura 4.4: Temps d'execució de la xarxa neuronal convolucional amb 16 canals.

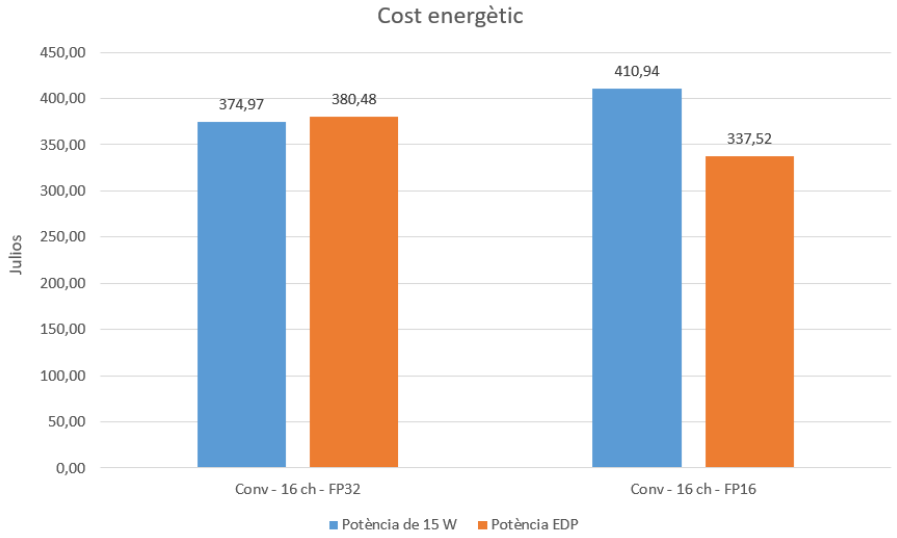

Figura 4.5: Consum energètic de la xarxa neuronal convolucional amb 16 canals.

- 6 capes convolucionals.
- 3 capes *maxpooling*.
- 2 capes completament connectades.
- 7 funcions d'activació ReLU.
- 4 capes *Dropout*.

La descripció i els paràmetres de cada capa d'aquesta xarxa, mostrats quan es llança a execució HELENNA, es poden veure a la Figura C.3 de l'apèndix C.

En la gràfica de la Figura 4.7 es poden observar els temps d'execució dels entrenaments amb la xarxa neuronal vgg3-dropout. El millor resultat que s'ha obtingut és el del mòdul cublas16 amb el mode de potència EDP. No obstant això, si comparem entre els mòduls, veiem que en el mode de potència de 15 W, cublas16 és 1% més ràpid. En el cas del mode EDP, ocorre una cosa similar, ja que cublas16 és un 12.58% més ràpid que

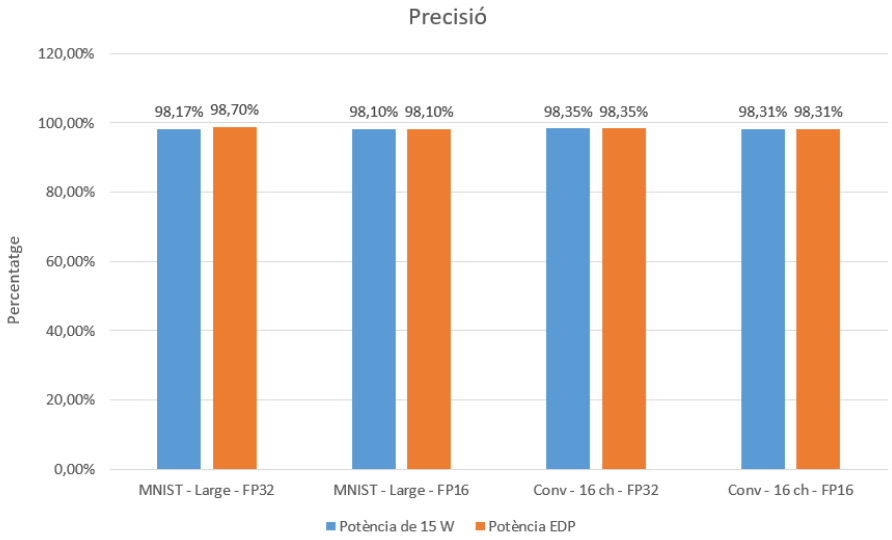

Figura 4.6: Precisió dels entrenaments amb les xarxes neuronals mlp_large i conv-16channels, les quals fan servir la base de dades MNIST.

cublas. Per altra banda, si comparem entre modes de potència, utilitzar el mode EDP representa una millora del 22.27% del temps d'execució. En el mòdul cublas16, EDP obté un 35.86% de millora en el temps de còmput.

Les dades que representen el cost energètic estan representades en la Figura 4.8, on es veu que l'opció que menys consumeix és el mòdul de cublas16 amb el mode EDP. Tanmateix, ens centrem en la diferenciació entre mòduls. En el mode de 15 W, cublas16 és el que menys consum energètic presenta, però tan sols es diferencia del mòdul cublas per un 3.76%. No obstant això, en el mode EDP la diferència entre els mòduls és més notòria, ja que cublas16 presenta un 24.19% de reducció del consum energètic. Si comparem els modes de potència, en el mòdul cublas, EDP consumeix un 11.48% menys, però en cublas16 es consumeix un 30.28% menys que en el mode de 15 W.

En l'última figura presentada (la Figura 4.9), s'exposa la precisió obtinguda en cada una de les execucions. En aquest cas, el mòdul cublas presenta la precisió més gran amb un 71,03% i la de cublas16 és una mica més menuda, 68.06%. En aquest tipus de xarxes neuronals no és tan senzill obtenir una alta precisió com en les xarxes neuronals visten anteriorment. Igualment, presenta una precisió assequible i en aquest cas, com en els altres, el mòdul cublas16 aconsegueix un bon resultat de precisió, el qual és quasi similar que cublas. Per tant, la millor configuració per executar aquesta xarxa és el mòdul cublas16 amb el mode de potència EDP, per ser el que menor temps i consum energètic presenta. Si es volguera utilitzar el mode de 15 W, el de cublas16 representaria la millor opció; encara que no hi ha molta diferència entre els dos mòduls, aquest segueix presentant uns valors inferiors.

4.4 Estudi de l'impacte de la mida de fils que hi ha en els blocs

Per finalitzar, s'ha realitzat un estudi per veure l'impacte que pot tenir el nombre de fil de cada bloc en cada xarxa neuronal implementada en HELENNA. Com que s'han llançat huit execucions per cada xarxa neuronal, tan sols s'han pogut obtenir les dades de quatre xarxes neuronals. Aquestes utilitzen totes la base de dades MNIST. En total s'han realitzat 32 execucions i les dades obtingudes s'han representat en la Taula 4.1. En aquesta taula es pot veure el nombre de fils, la precisió i el temps d'execució obtinguts en

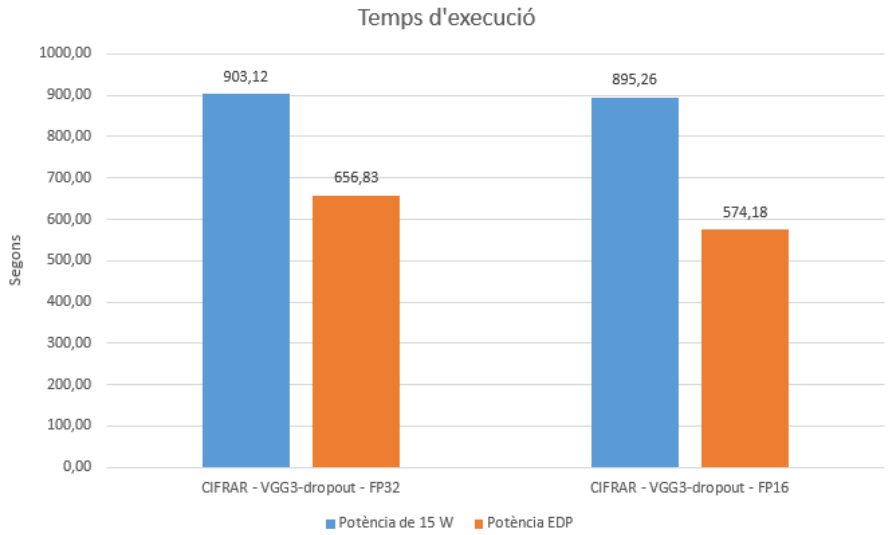

Figura 4.7: Temps d'execució de la xarxa neuronal vgg3-dropout.

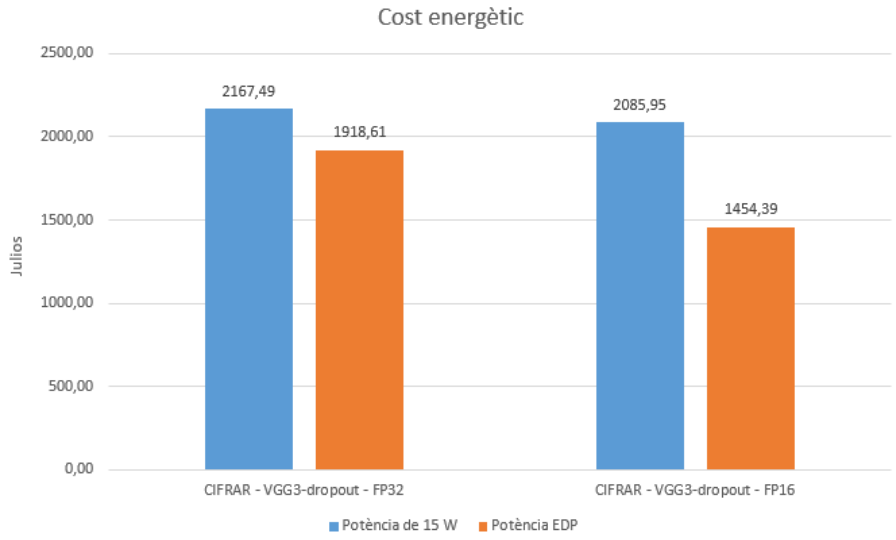

Figura 4.8: Consum energètic de la xarxa neuronal vgg3-dropout.

cada entrenament. Cal recalcar que aquestes execucions s'han dut a terme amb el mòdul cublas16 i la mateixa configuració descrita al principi del capítol, però només utilitzem una època, ja que en aquest cas obtenir una alta precisió no és important.

En primer lloc, posem el focus en la precisió. En totes les xarxes neuronals la precisió es manté estable a pesar de canviar la mida dels blocs. Açò és a causa del fet que aquests canvis no afecten en el procés de fer les operacions, l'únic que implica és que es farà d'una forma més o menys ràpida. Per tant, la diferencia de precisió entre les diferents execucions no són molt notòries. On si es veu un gran decrement, és quan s'assigna la mida 2049 i 4096, on no se supera ni el 10% de precisió. Açò és pel fet que el nombre màxim de fils que es pot tenir per bloc és de 1024, cosa que es va implantar a partir de l'arquitectura Volta i que es menciona en el capítol 2. Aleshores, quan s'assignen més de 1024 fils, la GPU no executa cap tasca. Açò es veu reflectit en les dues últimes files de la taula on clarament s'obté un temps molt inferior en comparació amb el temps d'execució de la xarxa neuronal.

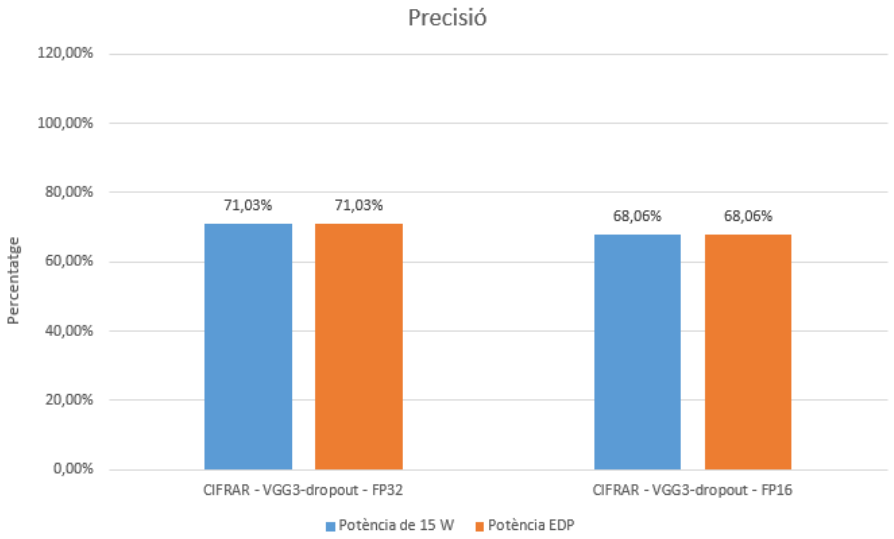

Figura 4.9: Precisió de l'entrenament amb vgg3-dropout fent ús d'HELENNA.

En segon lloc, si ens centrem en el temps d'execució, s'aprecia que, segons la grandària de la xarxa neuronal és millor una configuració o una altra. A continuació, detallem quines són les millors i pitjors configuracions en cada xarxa neuronal. Aquestes són:

- mlp_tiny: aquesta xarxa és la més menuda de totes i està formada per capes completament connectades. En aquesta el millor temps d'execució obtingut correspon a la mida de 256 fils i el pitjor resultat correspon a 32 fils per bloc.
- mlp_medium: aquesta és igual que l'anterior, però un poc més gran. En aquest cas el millor resultat s'obté amb 64 fils i la pitjor amb 128 fils.
- mlp_large: és la que s'ha utilitzat per a presentar les dades de les seccions anteriors. El millor temps d'execució és amb 128 fils i el pitjor amb 32 fils.
- conv: aquesta és la més menuda de totes les xarxes neuronals que tenen capes de convolució. El millor temps que s'aconsegueix és quan s'executa l'entrenament amb 512 fils i el pitjor és quan s'executa en 32 fils.

Com a conclusió d'aquesta anàlisi, es pot afirmar que quan es tinguen xarxes neuronals de gran mida, obtindran millors resultats si la mida dels blocs és gran, perquè com més gran siga el bloc més fils es podran executar a la vegada i s'accelerarà l'execució. Un exemple pot ser el cas de la xarxa mlp_large on la mida més menuda de fils és la que pitjors resultats mostra, però com més gran és la mida del bloc, millors resultats presenten. També, s'ha de recalcar que les diferències entre les diferents mides no són significatives.

	MLP_TINY		**MLP_MEDIUM**		**MLP_LARGE**		**CONV**	
Mida blocs	Precisió	T. exe.	Precisió	T. exe.	Precisió	T. exe.	Precisió	T. exe.
32	90.97	7.61	96.73	22.36	96.18	271.81	90.83	45.09
64	90.94	10.66	96.82	22.27	96.45	173.59	90.82	19.39
128	90.95	6.56	96.87	27.83	95.61	139.84	90.85	23.71
256	90.96	6.49	96.75	27.61	96.26	137.35	90.82	22.30
512	90.96	6.56	96.83	25.41	96.08	139.77	90.82	12.20
1024	90.96	6.52	96.83	27.51	95.61	141.10	90.82	21.43
2048	9.8	5.52	9.8	15.77	9.8	80.19	9.8	12.91
4096	9.8	7.66	9.8	15.69	9.8	81.74	9.8	8.13

Taula 4.1: Estudi de l'impacte del nombre de fils d'un bloc en l'execució d'una aplicació. L'estudi es realitza amb les xarxes neuronals de la base de dades MNIST i es presenta la precisió en percentatge i el temps d'execució en segons.

CAPÍTOL 5
Conclusions i treball futur

Per a finalitzar, en aquest capítol es detalla tot l'aconseguit al llarg del treball i de les possibles tasques per ampliar-lo en un futur.

En primer lloc, l'objectiu principal del treball és donar un suport ple a HELENNA per a executar xarxes neuronals amb nombres reals de mitja precisió. Per tal d'assolir aquest objectiu s'ha preparat HELENNA perquè puga tindre un nou mòdul, anomenat cublas16, i s'han implementat totes les funcions de les capes següents:

- La capa completament connectada: és la capa que més funcions té, aleshores, ha sigut una de les capes més costoses d'implementar. Amb aquesta capa s'ha obtingut una alta precisió similar al mòdul cublas amb un menor temps d'execució.
- La capa convolucional: les funcions d'aquesta són de comparació i assignació de valors. Els entrenaments amb aquesta capa també han obtingut una alta precisió, però el temps d'execució en algunes ocasions és similar o superior al mòdul cublas. Cosa que ens ha fet reduir el ritme de treball, ja que no es localitzava l'error exacte que provocava aquest comportament.
- La capa de normalització: existeixen dos tipus de capes, les quals són la *Batch Normalization* i la *Dropout*. En el cas de la *Batch Normalization*, s'implementaren totes les funcions, però no s'aconseguí una bona precisió en l'entrenament de xarxes neuronals que utilitzaven aquest tipus de capa. Degut aquest fet es va fer servir una cerca exhaustiva per a localitzar l'error exacte. D'altra banda, en la *Dropout* s'ha aconseguit una precisió i temps d'execució similar al que s'obté en el mòdul cublas.

Una vegada implementades totes les capes s'optimitzaren algunes funcions per reduir el seu cost temporal. Aquestes són: les operacions de conversió de valors entre nombres reals de simple i mitja precisió, la funció `matadd_col` i la `matset_random_ones`, la qual és la funció principal de la capa *Dropot*. L'anàlisi i explicació d'aquestes funcions es pot consultar en el capítol 3, secció 3.5.

A continuació, s'ha elaborat una anàlisi de les dades d'execució obtingudes on es comparen els resultats del mòdul cublas amb el mòdul cublas16 i es veu l'impacte energètic que té cada execució. A més, s'ha fet un estudi per veure la repercussió que té la grandària del bloc en diferents entrenaments de xarxes neuronals amb mides distintes. Aquestes anàlisis es poden consultar en el capítol 4.

Com a conclusions podem garantir que s'ha implementat un nou mòdul en HELENNA que suporta l'entrenament de xarxes neuronals formades per capes completament connectades, convolucionals i *Dropout*. A més, els resultats demostren que ofereixen una alta precisió i un cost energètic i temporal menor que el mòdul implementat amb l'aritmètica de coma flotant de 32 bits.

5.1 Treball futur

En aquesta secció s'aborden totes les tasques que es poden realitzar per completar el treball en el futur. Aquestes tasques són:

1. Optimització de la capa convolucional. Com que les operacions que es realitzen en aquesta capa són de comparació i assignació de valors, aleshores, no hauria de suposar un alt cost temporal, però les dades reflecteixen tot el contrari. Per tant, l'alt cost temporal dels entrenaments amb aquests tipus de xarxes neuronals no és per causa de la implementació en si, sinó per l'accés a memòria. Dit d'una altra forma, els accessos a memòria són els que realment augmenten el temps d'execució. Aleshores, cal implementar una altra versió de les funcions on es puga accedir a memòria d'una forma més eficient.

2. Canvi de la implementació de la capa *Batch Normalization*. En aquest cas, com que no és possible executar aquesta capa amb els nombres en coma flotant de 16 bits, s'ha de redissenyar la implementació realitzada i fer-la de manera híbrida. Per tal d'assolir-ho cal canviar totes les funcions que formen aquesta capa perquè facen les operacions amb nombres en coma flotant de 32 bits. Però, abans de cridar a aquestes funcions, s'ha de convertir els *buffers* que es passen com a paràmetres d'entrada, en nombres en coma flotant de 32 bits, ja que aquests *buffers* contindran nombres de mitja precisió.

3. Optimització d'altres funcions de la capa *fully connected*. En la capa completament connectada hi ha funcions que necessiten ser optimitzades a causa del seu alt cost temporal. Aleshores, s'hauria de continuar amb el treball d'optimització de les funcions a fi de obtenir uns millors resultats dels presentats en el capítol 4.

4. Continuació de l'estudi de l'impacte de la grandària del bloc sobre l'entrenament i inferència de les xarxes neuronals: la continuació d'aquest estudi consisteix a realitzar les diferents execucions, on es canvia la mida del bloc. Aquesta ampliació de l'estudi consisteix a fer els entrenaments amb xarxes neuronals que utilitzen capes convolucionals més grans i altres que facen servir la base de dades Cifar-10. Amb aquesta ampliació es pretén veure el comportament de l'entrenament amb xarxes més denses quan es canvia la grandària del bloc.

APÈNDIX A
Codi de la capa completament connectada

En aquest apèndix es mostra el codi de les dues funcions explicades en la secció 3.2 i la diferència entre utilitzar el *kernel* i la biblioteca cuBLAS. A continuació, es mostra el codi que s'ha utilitzat en la funció `matmul` descrita en el capítol 3.

Funció `matmul`

```
1  EXTERN_C int fn_matmul_cublas16(void *ptr_a, int rows_a, int cols_a,
2          void *ptr_b, int rows_b, int cols_b, void *ptr_c, int cols_c, int
3          offset_c)
4  {
5      //printf("fn_matmul not implemented yet\n"); exit(1);
6      PROFILING_HEADER_EXTERN(matmul);
7      PROFILING_DEVICE(matmul, DEV_CUBLAS16);
8      __half alf = __float2half(1.0f);
9      __half bet = __float2half(0.0f);
10     __half *aux_ptr_c = (__half *) ptr_c;
11
12     status16 = cublasHgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, cols_b,
13         rows_a, cols_a, &alf,(__half *) ptr_b, cols_b,(__half *) ptr_a,
14         cols_a, &bet, aux_ptr_c+offset_c, cols_c);
15
16     fn_synchronize16();
17     check_status16(status16, "fn_matmul_cublas16");
18     return 1;
19
20 }
```

En aquest codi es poden veure (línies 6 i 7) dues funcions d'HELENNA que serveixen per a obtenir el temps d'execució de cada funció i el seu temps total, al final de l'execució. Després, en les línies 8 i 9 es realitza la conversió d'un valor *float* a *half*. També, la crida a la funció `cublasHgemm()` de la biblioteca cuBLAS. Cal recalcar que aquesta és de les úniques funcions de la biblioteca on s'hi es pot utilitzar les dades de tipus *__half*. Aquesta funció té com a paràmetres d'entrada: les matrius, el nombre de columnes i files de cada una i la disposició de la matriu, és a dir, si la matriu s'ha de transposar (`CUBLAS_OP_T`) o no (`CUBLAS_OP_N`). Per últim, es fa una crida a la funció de sincronització (línia 16) per a sincronitzar CPU i GPU i una crida a `check_status16()` per tal de comprovar que la funció `cublasHgemm()` s'ha executat correctament.

Funció `matadd`

El codi de la funció `matadd` està implementat utilitzant el *kernel*, ja que no es pot utilitzar una funció de la biblioteca cuBLAS. La primera funció està en el fitxer `cublas16.cpp` i les altres dos estan en el fitxer `cublas16_kernels.cu`. El codi és el següent:

```
1         //Implementada en cublas16.cpp
2 EXTERN_C int fn_matadd_cublas16(void *a, int rows_a, int cols_a,
3         void *b, int rows_b, int cols_b, void *d)
4 {
5     //printf("fn_matadd not implemented yet\n"); exit(1);
6     PROFILING_HEADER_EXTERN(matadd);
7     PROFILING_DEVICE(matadd, DEV_CUBLAS16);
8
9     float al = 1.0f;
10    fn_haxpy_cublas16_kernel(al, a, b, cols_b, rows_b, d);
11    fn_synchronize16();
12    check_status16(status16, "fn_matadd_cublas16");
13    return 1;
14
15 }
16
17        //Implementades en cublas16_kernels.cu
18            //Codi executat en GPU
19 __global__ void fn_haxpy_device(float al, __half *x, __half *y,
20     __half *d, int n)
21 {
22     int tid = threadIdx.x + blockIdx.x * blockDim.x;
23     int stride = blockDim.x * gridDim.x;
24     __half fh = __float2half(al);
25     __half aux;
26     #if __CUDA_ARCH__ >= 530
27     for(int i = tid; i < n; i += stride){
28         aux = __hmul(fh, y[i]);
29         d[i] = __hadd(x[i], aux);
30     }
31     #endif
32 }
33
34            //Crida al kernel
35 void fn_haxpy_cublas16_kernel(float al, void *a, void *b, int cols_b,
36     int rows_b, void *d)
37 {
38     int n = cols_b * rows_b;
39     size_t num_threads = MAX_THREADS_PER_BLOCK;
40     size_t num_blocks = (n / num_threads) + ((n % num_threads) ? 1 : 0);
41     fn_haxpy_device<<<num_blocks, num_threads>>>(al, (__half *) a, (__half *) b
          , (__half *) d, n);
42     fn_synchronize_kernel();
43 }
```

La primera funció crida a la funció llançadora del fitxer `check_status16()` per a executar les operacions en la GPU (línia 10). De la línia 35 a la 43 tenim la funció llançadora. En aquesta es defineix el nombre de fils (línia 39) i el nombre de blocs (línia 40). En la línia 41 està ubicada la crida al *kernel* i en la 42 tenim la funció de sincronització. Després, de les línies 19 a la 32 tenim el codi que s'executa exclusivament en la GPU, on estan les variables per identificar el fil (`tid`) i el bloc (`stride`). En la 26 hi ha definit un *if* a través d'una macro per a indicar que, si l'arquitectura definida és major que 530, s'execute el codi, ja que les funcions aritmètiques de *half* funcionen quan l'arquitectura és igual o major que 530, l'AGX Xavier utilitza la 700. Per últim, dins del bucle que recorre les posicions del vector es realitza una multiplicació (línia 28) i una suma (línia 29).

APÈNDIX B
Codi optimitzat

En aquest apèndix es presenta el codi de l'operació de conversió i de les dues funcions explicades en la secció 3.5. Es presenta el codi inicial i l'optimització.

Operació de conversió

A continuació, es presenta el codi de les funcions `read_buffer` i `write_buffer`, implementades en el fitxer `cublas16.cpp`:

```
1
2  EXTERN_C int fn_read_buffer_cublas16(void *src, float *dst, size_t size, int
       keep_size)
3  {
4      //printf("fn_read_buffer_cublas16 not implemented yet\n"); exit(1);
5
6      size_t size_half;
7
8      if (keep_size == 0) {
9        size_half = size/2;
10         __half *h1_host = (__half *) malloc(size_half);
11       int tam = (size/sizeof(float));
12
13       mallocstatus = cudaMemcpy(h1_host,src,size_half,cudaMemcpyDeviceToHost);
14
15           //Calcular el temps de conversio
16       PROFILING_HEADER_EXTERN(conversion_read);
17       PROFILING_DEVICE(conversion_read, DEV_CUBLAS16);
18
19           //LES CONVERSIONS ES FAN EN CPU
20       #pragma omp parallel for
21       for(int i = 0; i < tam; i++){
22         dst[i] = __half2float(h1_host[i]);
23       }
24       //LES CONVERSIONS ES FAN EN GPU
25       /*void *f1_host = fn_allocate_buffer_cublas16(size, 1);
26       fn_synchronize16();
27       fn_conversion16_kernel(src, f1_host, tam, 0);
28       fn_synchronize16();
29       mallocstatus = cudaMemcpy(dst,f1_host,size,cudaMemcpyDeviceToHost);
30       fn_deallocate_buffer_cublas16(f1_host);*/
31
32       PROFILING_FOOTER(conversion_read);
33       free(h1_host);
34       check_malloc_status16(mallocstatus, "fn_read_buffer_cublas16_floats");
35
36     } else {
37         size_half = size;
```

```
38          mallocstatus = cudaMemcpy(dst, src, size_half, cudaMemcpyDeviceToHost);
39          check_malloc_status16(mallocstatus, "fn_read_buffer_cublas16_ints");
40      }
41      fn_synchronize16();
42      return 1;
43  }
44
45  EXTERN_C int fn_write_buffer_cublas16(float *src, void *dst, size_t size, int
        keep_size)
46  {
47      //printf("fn_write_buffer_cublas16 not implemented yet\n"); exit(1);
48
49      size_t size_half;
50      if (keep_size == 0) {
51
52        size_half = size/2;
53        int tam = (size/sizeof(float));
54        __half *h1_host = (__half *) malloc(size_half);
55
56        PROFILING_HEADER_EXTERN(conversion_write);
57        PROFILING_DEVICE(conversion_write, DEV_CUBLAS16);
58
59        //LES CONVERSIONS ES FAN EN CPU
60        #pragma omp parallel for
61        for(int i = 0; i < tam; i++){
62              h1_host[i] = __float2half(src[i]);
63          }
64              //LES CONVERSIONS ES FAN EN GPU
65          /*void *f1_host = fn_allocate_buffer_cublas16(size, 1);
66          mallocstatus = cudaMemcpy(f1_host , src, size, cudaMemcpyHostToDevice);
67          fn_synchronize16();
68          fn_conversion16_kernel(dst, f1_host, tam, 1);
69          fn_synchronize16();
70          fn_deallocate_buffer_cublas16(f1_host);*/
71
72        PROFILING_FOOTER(conversion_write);
73        mallocstatus = cudaMemcpy(dst,h1_host,size_half,cudaMemcpyHostToDevice);
74        free(h1_host);
75        check_malloc_status16(mallocstatus, "fn_write_buffer_cublas16");
76
77      } else {
78           size_half = size;
79           mallocstatus = cudaMemcpy(dst, src, size_half,
80              cudaMemcpyHostToDevice);
81           check_malloc_status16(mallocstatus, "fn_write_buffer_cublas16");
82      }
83      fn_synchronize16();
84      return 1;
85  }
```

Aquest codi es compon de dues funcions, l'una de lectura i l'altra d'escriptura, però nosaltres les analitzarem en conjunt, ja que el codi en les dues és molt similar. En aquest codi es pot veure com segons si el *buffer* cal dividir-lo o no, s'executa un codi o un altre (línies 7 i 50). Les línies 20 a la 23 i de la 60 a la 63 s'encarrega de fer la conversió entre valors, on s'ha posat la macro d'OpenMP per a paral·lelitzar el bucle. Si s'observen les línies 24 a la 30 i de la 64 a la 70 tenim el codi corresponent a l'execució de la conversió en GPU. En aquest cas es cridava a una funció llançadora i després al *kernel*. Com que aquesta operació es feia en GPU, era necessari un *buffer* auxiliar assignat en la seua memòria, passar-li les dades a convertir, fer la conversió i tornar a passar les dades convertides a CPU. Amb aquesta implementació es realitzaven dues transferències de dades

i una assignació de memòria, la qual cosa feia que el cost temporal fóra major. Per això, es decidí agafar la implementació inicial i paral·lelitzar-la amb OpenMP.

Funció `matadd_col`

La funció `matadd_col` està implementada en la GPU. Per tant, el codi que es mostra a continuació està implementat en el fitxer `cublas16.cpp` i `cublas16_kernels.cu`:

```
        //Implementada en cublas16.cpp
EXTERN_C int fn_matadd_col_cublas16(void *ptr_a, int rows_a, int cols_a, void *ptr_v)
{
    //printf("fn_matadd_col not implemented yet\n"); exit(1);

    PROFILING_HEADER_EXTERN(matadd_col);
    PROFILING_DEVICE(matadd_col, DEV_CUBLAS16);

    fn_matadd_col_kernel(ptr_a, rows_a, cols_a, ptr_v);
    fn_synchronize16();
    check_status16(status16, "fn_matadd_col_cublas16");
    return 1;
}

        //Implementades en cublas16_kernels.cu
            //Codi executat en GPU
__global__ void fn_matadd_col_device(__half *ptr_a, int rows_a, int cols_a, __half *ptr_v){

    //NO OPTIM
    /*int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
    #if __CUDA_ARCH__ >= 530
    if(tid<rows_a){
        for(int j=0; j<cols_a; j++){
            ptr_a[(tid*cols_a)+j] = __hadd(ptr_a[(tid*cols_a)+j],ptr_v[tid]);
        }
    }
  #endif*/

  int col = threadIdx.x;
  int row = blockIdx.x;
  int addr;

  #if __CUDA_ARCH__ >= 530
  while (col < cols_a){
      addr = (row * cols_a) + col;
      ptr_a[addr] = __hadd(ptr_a[addr], ptr_v[row]);
      //ptr_v te tants elements com files te ptr_a (blocs)
      col += MAX_THREADS_PER_BLOCK;
  }
  #endif
}

        //Codi que crida al kernel
void fn_matadd_col_kernel(void *ptr_a, int rows_a, int cols_a, void *ptr_v){
    int n = rows_a * cols_a;
    size_t num_threads = MAX_THREADS_PER_BLOCK; \\1024
    size_t num_blocks = (n / num_threads) + ((n % num_threads) ? 1 : 0);
    //size_t num_blocks = rows_a;
    fn_matadd_col_device<<<num_blocks, num_threads>>>((__half *) ptr_a, rows_a,
         cols_a, (__half *) ptr_v);
    fn_synchronize_kernel();
}
```

El *kernel* és la funció principal de les tres, ja que implementa l'operació aritmètica de la funció. El primer codi que es va fer, el no optimitzat, es pot veure a des de les línies 19 a la 28. Aquest codi consta d'un `for` que recorre els índexs dels *buffers* i es fa una suma dels valors. Aquest `for` no és òptim, perquè es podia donar el cas on diferents fils podien fer mateixa operació amb els mateixos índex de memòria, augmentant el cost temporal. Amb la nova implementació (línies 30 – 42) tan sols s'agafen els índexs del fil i bloc. Amb el *while* es recorre cada posició de memòria que li correspon al fil, aleshores cada fil accedeix a les seues posicions i no es produeixen més accessos dels necessaris.

Funció `matset_random_ones`

La funció `matset_random_ones` està implementada únicament en el fitxer `cublas16.cpp`. Aquest codi és:

```
1  EXTERN_C int fn_random_ones_cublas16(void *ptr_m, int rows, int cols, float
       prob) {
2      //printf("fn_matset_random_ones not implemented yet\n"); //exit(1);
3      PROFILING_HEADER_EXTERN(mat_set);
4      PROFILING_DEVICE(mat_set, DEV_CUBLAS16);
5
6      size_t size = rows*cols*sizeof(float);
7
8          //cuRAND
9      float* gpu_m;
10     mallocstatus = cudaMalloc(&gpu_m, size);
11     check_malloc_status16(mallocstatus, "allocated matset");
12     curandGenerateUniform(gen, gpu_m, rows*cols);
13     fn_synchronize16();
14     fn_conversion16_kernel(ptr_m, gpu_m, rows*cols, 1);
15     fn_synchronize16();
16
17         //CPU
18     /*float *cpu_m = (float *) malloc(size);
19     for(int i=0; i<rows; i++){
20      for(int j=0; j<cols; j++){
21        cpu_m[i*cols+j]=static_cast<float>(rand())/static_cast<float>(RAND_MAX);
22       }
23     }
24     fn_write_buffer_cublas16(cpu_m, ptr_m, size, 0);*/
25
26     fn_matset_random_ones_cublas16_kernel(ptr_m, rows, cols, prob);
27     cudaFree(gpu_m);
28     //free(cpu_m);
29     fn_synchronize16();
30     return 1;
31 }
```

Aquesta funció inicialment es va implementar per a executar-se en la CPU (línies 18 – 24, 28), on es generen els nombres aleatoris i després es passen els valors a GPU. D'altra banda, les linies 9 – 14 mostren com s'ha implementat la generació dels nombres amb la biblioteca cuRAND. La funció d'aquesta biblioteca és la mostrada en la línia 12, on guarda els valors en un *buffer* auxiliar. Després, amb la crida de la funció de conversió, es converteixen els valors a *half* igual que en les funcions de *read* i *write*. L'única diferència és que en aquesta funció la conversió es fa entre vectors de GPU, aleshores no cal transferir els valors de CPU a GPU.

APÈNDIX C
Topologies de les xarxes neuronals

En aquest apèndix es mostren les topologies que formen les xarxes neuronals utilitzades per a les anàlisis de resultats del capítol 4. Les figures següents mostren el nom de les capes, el tipus de capa, el nombre de neurones, els paràmetres que tenen cada una d'elles i la seua configuració.

network and model summary

layer	name	layer type	neurons	params	in_layer	Format	configur	memory
0	input	input layer	784	0		IHWB	inputs 784, outputs 784	0.00 GB
1	fc1_0	fully connected	2000	1570000	(prev)	IHWB	inputs 784, outputs 2000	0.03 GB
2	relu1_0	relu	2000	0	(prev)	IHWB		0.00 GB
3	fc1_0	fully connected	2000	4002000	(prev)	IHWB	inputs 2000, outputs 2000	0.07 GB
4	relu1_0	relu	2000	0	(prev)	IHWB		0.00 GB
5	fc1_0	fully connected	2000	4002000	(prev)	IHWB	inputs 2000, outputs 2000	0.07 GB
6	relu1_0	relu	2000	0	(prev)	IHWB		0.00 GB
7	fc1_0	fully connected	2000	4002000	(prev)	IHWB	inputs 2000, outputs 2000	0.07 GB
8	relu1_0	relu	2000	0	(prev)	IHWB		0.00 GB
9	fc1_0	fully connected	2000	4002000	(prev)	IHWB	inputs 2000, outputs 2000	0.07 GB
10	relu1_0	relu	2000	0	(prev)	IHWB		0.00 GB
11	fc1_0	fully connected	2000	4002000	(prev)	IHWB	inputs 2000, outputs 2000	0.07 GB
12	relu1_0	relu	2000	0	(prev)	IHWB		0.00 GB
13	fc1_0	fully connected	2000	4002000	(prev)	IHWB	inputs 2000, outputs 2000	0.07 GB
14	relu1_0	relu	2000	0	(prev)	IHWB		0.00 GB
15	fc1_0	fully connected	2000	4002000	(prev)	IHWB	inputs 2000, outputs 2000	0.07 GB
16	relu1_0	relu	2000	0	(prev)	IHWB		0.00 GB
17	fc1_0	fully connected	2000	4002000	(prev)	IHWB	inputs 2000, outputs 2000	0.07 GB
18	relu1_0	relu	2000	0	(prev)	IHWB		0.00 GB
19	fc1_0	fully connected	10	20010	(prev)	IHWB	inputs 2000, outputs 10	0.00 GB
20	output_0	softmax	10	0	(prev)	IHWB		0.00 GB
		TOTAL	36020	33606010		Memory (no layers):	0.00 GB	0.63 GB

Figura C.1: Topologia de la xarxa neuronal mlp_large.

network and model summary

layer	name	layer type	neurons	params	in_layer	Format	configuration	memory
0	input	input layer	784	0		IHWB	inputs 784, outputs 784	0.00 GB
1	conv1_0	convolutional	12544	160	(prev)	IHWB	IN:1x28x28 KERNEL:3x3 PADDING:1x1 STRIDE:1x1 OUT:16x28x28	0.00 GB
2	relu1_0	relu	12544	0	(prev)	IHWB		0.00 GB
3	maxp1_0	maxpooling	3136	0	(prev)	IHWB	IN:16x28x28 KERNEL 2x2 PADDING:0x0 STRIDE 2x2 OUT:16x14x14	0.00 GB
4	fc2_0	fully connected	10	31370	(prev)	IHWB	inputs 3136, outputs 10	0.00 GB
5	softmax_0	softmax	10	0	(prev)	IHWB		0.00 GB
		TOTAL	28244	31530		Memory (no layers):	0.00 GB	0.01 GB

Figura C.2: Topologia de la xarxa neuronal conv-16channels.

network and model summary							
layer	name	layer type	neurons	params	in_layer	Format	configuration	memory
0	input	input layer	3072	0		IHWB	inputs 3072, outputs 3072	0.00 GB
1	conv1_0	convolutional	32768	896	(prev)	IHWB	IN:3x32x32 KERNEL:3x3 PADDING:1x1 STRIDE:1x1 OUT:32x32x32	0.01 GB
2	relu1_0	relu	32768	0	(prev)	IHWB		0.01 GB
3	conv2_0	convolutional	32768	9248	(prev)	IHWB	IN:32x32x32 KERNEL:3x3 PADDING:1x1 STRIDE:1x1 OUT:32x32x32	0.04 GB
4	relu2_0	relu	32768	0	(prev)	IHWB		0.01 GB
5	maxp1_0	maxpooling	8192	0	(prev)	IHWB	IN:32x32x32 KERNEL 2x2 PADDING:0x0 STRIDE 2x2 OUT:32x16x16	0.00 GB
6	dropout1_0	dropout	8192	0	(prev)	IHWB	RATE: 0.200000	0.00 GB
7	conv4_0	convolutional	16384	18496	(prev)	IHWB	IN:32x16x16 KERNEL:3x3 PADDING:1x1 STRIDE:1x1 OUT:64x16x16	0.01 GB
8	relu3_0	relu	16384	0	(prev)	IHWB		0.00 GB
9	conv4_0	convolutional	16384	36928	(prev)	IHWB	IN:64x16x16 KERNEL:3x3 PADDING:1x1 STRIDE:1x1 OUT:64x16x16	0.02 GB
10	relu4_0	relu	16384	0	(prev)	IHWB		0.00 GB
11	maxp2_0	maxpooling	4096	0	(prev)	IHWB	IN:64x16x16 KERNEL 2x2 PADDING:0x0 STRIDE 2x2 OUT:64x8x8	0.00 GB
12	dropout2_0	dropout	4096	0	(prev)	IHWB	RATE: 0.200000	0.00 GB
13	conv5_0	convolutional	8192	73856	(prev)	IHWB	IN:64x8x8 KERNEL:3x3 PADDING:1x1 STRIDE:1x1 OUT:128x8x8	0.01 GB
14	relu5_0	relu	8192	0	(prev)	IHWB		0.00 GB
15	conv6_0	convolutional	8192	147584	(prev)	IHWB	IN:128x8x8 KERNEL:3x3 PADDING:1x1 STRIDE:1x1 OUT:128x8x8	0.01 GB
16	relu6_0	relu	8192	0	(prev)	IHWB		0.00 GB
17	maxp3_0	maxpooling	2048	0	(prev)	IHWB	IN:128x8x8 KERNEL 2x2 PADDING:0x0 STRIDE 2x2 OUT:128x4x4	0.00 GB
18	dropout3_0	dropout	2048	0	(prev)	IHWB	RATE: 0.200000	0.00 GB
19	fc1_0	fully connected	128	262272	(prev)	IHWB	inputs 2048, outputs 128	0.00 GB
20	relu7_0	relu	128	0	(prev)	IHWB		0.00 GB
21	dropout4_0	dropout	128	0	(prev)	IHWB	RATE: 0.200000	0.00 GB
22	fc2_0	fully connected	10	1290	(prev)	IHWB	inputs 128, outputs 10	0.00 GB
23	softmax_0	softmax	10	0	(prev)	IHWB		0.00 GB
		TOTAL	258452	550570			Memory (no layers): 0.00 GB	0.14 GB

Figura C.3: Topologia de la xarxa neuronal vgg3-dropout.

Bibliografia

[1] John Cheng, Max Grossman, Ty McKercher. *CUDA C Programming*. John Wiley & Sons, Inc., Hoboken, 2014.

[2] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPU*. Morgan Kaufmann, Burlington, 2018.

[3] CPU vs GPU: ¿cuál es la diferencia? Consultat a https://www.intel.es/content/www/es/es/products/docs/processors/cpu-vs-gpu.html, el 14 de juny del 2021.

[4] Famous Graphics Chips: Intel's GPU History. Publicat per Dr. Jon Peddie. Consultat a https://www.computer.org/publications/tech-news/chasing-pixels/intels-gpu-history, el 14 de juny del 2021.

[5] Famous Graphics Chips: NEC µPD7220 Graphics Display Controller. Publicat per Dr. Jon Peddie. Consultat a https://www.computer.org/publications/tech-news/chasing-pixels/famous-graphics-chips, el 14 de juny del 2021.

[6] Jaegeun Han, Bharatkumar Sharma. *Learn CUDA Programming: A beginner's guide to GPU programming and parallel computing with CUDA 10.x and C/C++*. Packt Publishing Ltd, Birmingham, 2019.

[7] Jeff Heaton. *Artificial Intelligence for Humans, Volume 3: Deep Learning and Neural Networks*. Heaton Research, Inc, Chesterfield, primera edició, 2015.

[8] John L. Hennessy, David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Burlington, sisena edició, 2019.

[9] Història de NVIDIA. Consultat a https://www.nvidia.com/es-es/about-nvidia/corporate-timeline/, el 15 de juny del 2021.

[10] William Hohl, Christopher Hinds. *ARM Assembly Language: Fundamentals and Techniques*. CRC Press, Boca Raton, Segona Edició, 2015.

[11] Informació sobre la Nintendo 64. Consultat a https://www.nintendo.es/Empresa/La-historia-de-Nintendo/Nintendo-64/Nintendo-64-625959.html, el 23 de juny del 2021.

[12] Is it Time to Rename the GPU? Publicat per Dr. Jon Peddie. Consultat a https://www.computer.org/publications/tech-news/chasing-pixels/is-it-time-to-rename-the-gpu, el 14 de juny del 2021.

[13] David B. Kirk i Wen-mei W. Hwu. *Programming Massively Parallel Processors*. Morgan Kaufmann, Burlington, segona edició, 2013.

[14] Modes de potència del dispositiu NVIDIA Jetson AGX Xavier. Consultat a https://developer.ridgerun.com/wiki/index.php?title=Xavier/JetPack_4.1/Performance_Tuning/Tuning_Power, el 20 de juny del 2021.

[15] Marc Moreno Lopez, Jugal Kalita. Deep Learning applied to NLP. *arXiv preprint arXiv:1703.03091*, 9 Mar 2017.

[16] Neural Networks, emés el 17 d'agost del 2020. Publicat per IBM Cloud Education. Consultat a https://www.ibm.com/cloud/learn/neural-networks#toc-history-of-rIfu5uF2, el 24 de juny del 2021.

[17] NVIDIA. cuBlas Library: User Guide, emés en setembre del 2020. Consultat a https://docs.NVIDIA.com/pdf/CUBLAS_Library.pdf, el 21 de juny del 2021.

[18] NVIDIA. cuRAND Library: Programming Guide, emés en abril del 2021. Consultat a https://docs.nvidia.com/cuda/pdf/CURAND_Library.pdf, el 21 de juny del 2021.

[19] NVIDIA. Cuda-memcheck: User Manual, emés en abril de 2021. Consultat a https://docs.NVIDIA.com/cuda/pdf/CUDA_Memcheck.pdf, el 22 de juny del 2021.

[20] NVIDIA. Cuda C++ Best Practices Guide: Design Guide, emés en juny de 2021. Consultat a https://docs.NVIDIA.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf, el 26 de juny del 2021.

[21] NVIDIA. CUDA Math API: API Reference Manual, emés en abril de 2021. Consultat a https://docs.NVIDIA.com/cuda/pdf/CUDA_Math_API.pdf, 23 de juny del 2021.

[22] NVIDIA. Profiler: User's Guide, emés en abril de 2021. Consultat a https://docs.NVIDIA.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf, el 22 de juny del 2021.

[23] NVIDIA. Tegrastats Utility, emés en juny de 2019. Consultat a https://docs.NVIDIA.com/drive/drive_os_5.1.6.1L/nvvib_docs/index.html#page/DRIVE_OS_Linux_SDK_Development_Guide/Utilities/util_tegrastats.html#wwpID0E0HB0HA, el 22 de juny del 2021.

[24] NVIDIA Developer. Caractresitiques del dispositiu NVIDIA Jetson AGX Xavier, emés el 12 de decembre de 2018. Consultat a https://developer.NVIDIA.com/blog/NVIDIA-jetson-agx-xavier-32-teraops-ai-robotics/?ncid=so-fac-mdjngxxrmllhml-69163, el 16 de juny del 2021.

[25] NVIDIA Developer. Mixed-Precision Programming with CUDA 8, emés el 19 d'octubre de 2016. Consultat a https://developer.NVIDIA.com/blog/mixed-precision-programming-cuda-8/, el 24 de juny del 2021.

[26] Michael L. Overton. *Numerical Computing with IEEE Floating Point Arithmetic*. Society for Industrial and Applied Mathematics, Philadelphia, 2001.

[27] Performance speedup: Jetson TX2 vs AGX Xavier. Publicat per Fyodor Serzhenko. Consultat a https://www.fastcompression.com/blog/xavier-vs-tx2.htm, el 19 de juny del 2021.

[28] Jason Sanders, Edward Kandrot. *CUDA by Example: An introduction to general-purpose GPU programming*. Addison-Wesley, Boston, 2011.

[29] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, juny, 2014.

[30] Duane Storti, Mete Yurtoglu. *CUDA for Engineers: An Introduction to High-Performance Parallel Computing*. Addison-Wesley, Boston, 2016.

[31] V. Thakkar, S. Tewary i C. Chakraborty. Batch Normalization in Convolutional Neural Networks — A comparative study with CIFAR-10 data. *Fifth International Conference on Emerging Applications of Information Technology (EAIT)*, 2018, pp. 1-5, doi: 10.1109/EAIT.2018.8470438.

[32] The History of the Modern Graphics Processor. Publicat per Graham Singer el 7 de gener del 2021. Consultat a https://www.techspot.com/article/650-history-of-the-gpu/, el 14 de juny del 2021.

[33] Nicholas Wilt. *The CUDA Hanbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley, Boston, 2013.

[34] What's the Difference Between a CPU and a GPU? Publicat per Brian Claudfield el 16 de desembre del 2019. Consultat a https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/, el 14 de juny del 2021.

[35] Dan Zuras, et al. *IEEE Standard for Floating-Point Arithmetic*. IEEE Computer Society, New York, 2008.