Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN ENXEÑARÍA DE COMPUTADORES

Implementación en CUDA dun método para realizar a operación de convolución en lotes

Estudante: Sara Aguado Couselo
Dirección: Diego Andrade Canosa

A Coruña, setembro de 2020.

A todas as persoas que se interesaron por este proxecto

Agradecementos

Este proxecto vai dedicado a todo o que depositou nel o seu interese. Á miña familia, en especial a meus avós, por todos estes anos; e a Isaac, por brindar sempre o seu apoio. Tamén aos meus compañeiros e amigos da facultade, cos que coincidín neste camiño ás veces arduo e ás veces agradecido. Sen esquecer o traballo do meu titor dirixindo este proxecto e o de todos os profesores que souberon transmitir o seu coñecemento, e que recordaremos con agrado no futuro. Grazas tamén á xente do departamento, que me acollestes os primeiros meses, sobre todo a Roberto pola axuda prestada cos equipos e o material. Grazas a todos.

Resumo

Nos últimos anos, as plataformas heteroxéneas, tales como as tarxetas gráficas (GPU), tiveron un gran auxe na resolución de problemas en diversos ámbitos. A realización de operacións alxébricas por lotes xa foi explorada con éxito no pasado, como forma de mellorar o rendemento desta clase de operacións. Non obstante, existen diversas formas de realizalo. Algunhas intentan buscar un emprazamento óptimo das estruturas de datos en memoria, de forma que favoreza as características da plataforma na que o código será executado. Outras tratan de realizar un reparto do traballo que aumente a reutilización dos datos procesados por un mesmo fío. O proxecto explora todas estas estratexias, no marco dunha implementación que emprega CUDA para executar a operación de convolución por lotes. Esta operación alxébrica, ademais, é a que ocupa un maior tempo de execución no adestramento de redes de aprendizaxe profunda. Polo tanto, analizaremos o rendemento da implementación tanto de forma illada coma no contexto das redes de aprendizaxe profunda.

Abstract

In recent years, heterogeneous platforms (e.g., Graphical Processing Units), had a great boom solving problems in different fields. Batch algebraic operations have been successfully explored in the past as a way to improve performance. However, there are several ways to approach it. Some of them try to find an optimal location of the data structures in memory, in a way that favors the characteristics of the platform where the code is going to be executed. Others try to make a division of work that increases the reuse of data processed by the same thread. This project explores all of these strategies, as part of an implementation using CUDA to run the batched convolution operation. This algebraic operation is also the longest running operation in deep learning network training. Therefore, we will analyze implementation performance both in isolation and in the context of deep learning networks.

Palabras chave:

- GPGPU (Computación de Propósito Xeral en Unidades de Procesamento Gráfico)
- NVIDIA CUDA®
- Convolución por lotes

Keywords:

- GPGPU (General-Purpose Computing on Graphics Processing Units)
- NVIDIA CUDA®
- Batched Convolution

Índice Xeral

Índice de Figuras

Índice de Táboas

Capítulo 1

Introdución

Nos últimos anos, as plataformas heteroxéneas, aquelas que mesturan máis dun tipo de procesador ou núcleo, tiveron un gran auxe na resolución de problemas en ámbitos ata entón reservados para a CPU. Debido á especialización dos chips, a utilización dun par CPU-GPU proporciona vantaxes que non conseguiríamos usando múltiples CPUs en paralelo. Esta nova forma de traballar coas GPUs denomínase GPGPU (do inglés, *General-Purpose computing on Graphics Processing Units*).

Unha GPU é un procesador programable especializado no procesamento paralelo de grandes estruturas de datos. Aínda que foron deseñadas orixinalmente co propósito de renderizar imaxes, o certo é que a súa arquitectura, esencialmente SIMD (do inglés, *Single Instruction Multiple Data*), vólveas especialmente axeitadas para resolver problemas que consistan en aplicar unha mesma operación a múltiples datos. Se nos fixamos no esquema da figura 1.1 podemos ver as diferencias a nivel de arquitectura entre unha CPU e unha GPU. Por un lado, as primeiras están pensadas para reducir a latencia, o tempo que tarda en executarse unha única instrución. Para iso, as CPUs contan cunha unidade de control máis complexa e habitualmente incorporan mecanismos como a predicción de salto, cuxo obxectivo é anticiparse aos acontecementos para aforrar ciclos na execución de instrucións. Seguindo a mesma liña, dispoñen de unidades aritmético-lóxicas (ALU, do inglés *Arithmetic Logic Unit*) en menor cantidade pero de moita potencia, e unha caché de maior tamaño para reducir a latencia dos accesos a memoria. Por outro lado, as GPUs o que buscan é incrementar o rendemento xeral das operacións sobre grandes volumes de datos. Para logralo, utilizan moitos máis núcleos pero evitan implementar mecanismos de control excesivamente complexos. Estes núcleos empregan cachés máis pequenas e moitas máis ALUs, pero de maior latencia, a cambio dispoñen dunha gran capacidade de segmentación para un maior rendemento.

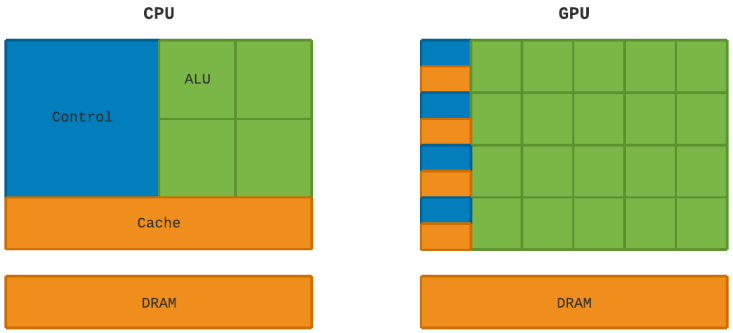


Figura 1.1: Arquitectura da CPU en contraposición coa da GPU

Podemos afirmar que a gran vantaxe das plataformas heteroxéneas é que, ao aproveitar as peculiaridades de ambos dispositivos, conseguen mellorar o rendemento dos programas empregando a CPU naquelas partes secuenciais onde o que importa é a latencia, e deixándolle o código paralelizable á GPU, que pode chegar a ser moito máis rápida resolvendo problemas con alto paralelismo de datos. Neste eido se sitúan as operacións de álxebra lineal, no campo que nos abrangue, poderíamos dicir que todas aquelas operacións que se aplican a vectores ou matrices. Este tipo de operacións ten unha gran cantidade de aplicacións en diversos campos, destacando no da informática: o modelado de gráficos por computador, o procesamento de imaxes e, nos últimos tempos, tamén a intelixencia artificial.

Pensemos, por exemplo, nos sistemas de visión artificial. Na súa operación estes sistemas procesan unha gran cantidade de imaxes, e moitos deles son aplicacións en tempo real, polo que o tempo de resposta debe ser mínimo. Cada imaxe almacénase en memoria como unha matriz de grandes dimensións, para facernos unha idea, unha imaxe en cor sen compresión e en *Full HD* (1920 x 1080 píxeles) ocupa case 25 MB, e sobre cada píxel de dita imaxe poden chegar a executarse múltiples operacións. Unha operación alxébrica recorrente é a convolución, trátase da operación que máis tempo ocupa no adestramento de redes de aprendizaxe profunda e sobre a que se vertebran as redes neuronais convolucionais. Sobre estas últimas falaremos máis en profundidade no segundo capítulo desta memoria. Por estas razóns, e polo gran protagonismo do que gozan actualmente as aplicacións que integran solucións baseadas na intelixencia artificial, implementaremos e analizaremos o rendemento da nosa aproximación tanto de forma illada coma no contexto das redes de aprendizaxe profunda.

Por outro lado, a realización de operacións alxébricas por lotes (en inglés, *batch*) foi explorada con éxito no pasado, como forma de mellorar o rendemento desta clase de operacións. Non obstante, existen diversas formas de abordar este problema. Algunhas intentan buscar un

emprazamento óptimo das estruturas de datos en memoria, de forma que favoreza as características da plataforma na que o código será executado. Outras tratan de realizar un reparto do traballo que aumente a reutilización dos datos procesados por un mesmo fío. O proxecto explora todas estas estratexias, no marco dunha implementación que emprega CUDA para executar a operación de convolución por lotes nunha GPU.

Finalmente, unha vez definido o problema e a forma de abordalo, só queda por determinar a ferramenta que imos utilizar na súa resolución. Á hora de aumentar a velocidade das nosas aplicacións e dependendo das nosas necesidades, o tempo do que dispoñemos, a meta que queiramos acadar, etc. empregaremos un tipo ou outro. En xeral, podemos definir tres tipos de ferramentas: en primeiro lugar, as librarías poden chegar a proporcionar un gran rendemento sen perdermos demasiado tempo en aprender a usalas. Nesta área temos o conxunto de librarías de NVIDIA[1], que abranguen distintos ámbitos: librarías matemáticas, especializadas en álxebra lineal, *Deep Learning*, procesamento de imaxe e vídeo, etc. En segundo lugar, as directivas de compilador tampouco requiren moito tempo de aprendizaxe e resultan máis portables, ao non ser específicas de ningún *hardware*, aínda que o seu rendemento varía en función da versión do compilador empregado. Os estándares OpenACC (*Open Accelerators*) e OpenMP (*Open Multi-Processing*)[2] simplifican a aceleración dos nosos programas en plataformas heteroxéneas, permitíndonos aplicar optimizacións e paralelizar o código en CPUs, GPGPUs e, no caso de OpenMP, tamén FPGAs. En terceiro e último lugar, as linguaxes de programación, aínda que moi ligadas ao *hardware*, proporcionan o maior rendemento á hora de programar plataformas heteroxéneas. Normalmente son máis complicadas de utilizar, pois o programador debe expresarse a un alto nivel de detalle, mais a flexibilidade e a capacidade de control que ofrecen, ao non estar limitadas a un conxunto de funcións como as dunha libraría, en certas aplicacións compensa o esforzo da formación inicial.

As primeiras linguaxes de programación en GPUs foron deseñadas para o procesamento de gráficos, actualmente existen varias implementacións que se centran na programación de propósito xeral, sendo OpenCL[3] a linguaxe predominante. Mais, tres anos antes do nacemento de OpenCL, NVIDIA publicaba a primeira versión de CUDA, unha linguaxe de programación similar a C/C++, que permitía mudar os conceptos gráficos por uns máis familiares no eido da computación de altas prestacións (HPC, do inglés *High-Performance Computing*). Como víamos adiantando, esta última foi a opción escollida neste traballo para realizar a nosa aproximación da convolución por lotes.

[1] GPU-Accelerated Libraries, dispoñibles no enderezo https://developer.nvidia.com/gpu-accelerated-libraries
[2] OpenACC (https://www.openacc.org/) e OpenMP (https://www.openmp.org/), páxinas oficiais.
[3] OpenCL, páxina oficial: https://www.khronos.org/opencl/

1.1 Obxectivos

Os obxectivos principais deste proxecto son os seguintes:

- **Entender como funciona e as vantaxes da operación de convolución por lotes**: Para realizar un proxecto de calquera tipo necesitamos adquirir certos coñecementos sobre a materia da que trata. O obxectivo principal deste proxecto é ampliar o noso horizonte de coñecementos e buscar tamén novas formas de abordar os problemas que se formulen no camiño.
- **Propoñer variantes para realizar o procesado por lotes da convolución**: Tendo en conta toda a información recollida, estudaranse as posibles alternativas, valorarase o seu rendemento e, a partir de aí, propoñeranse melloras ou versións alternativas.
- **Implementar en CUDA as variantes propostas**: Ademais de elaborar os algoritmos, é necesario realizar un estudo previo da linguaxe de programación ou das funcionalidades da mesma que se van empregar. Tamén é obrigado documentar o traballo realizado e manter un control de versións cunha ferramenta como pode ser *git*.
- **Analizar e comparar o rendemento da proposta con outras implementacións**: Co propósito de ir mellorando o rendemento en cada versión do algoritmo, realizarase un perfil do programa coas ferramentas que ofrece a plataforma e, a partir deste, unha análise dos resultados, que se utilizará para dirixir a busca dunha posible mellora.

1.2 Organización do proxecto

Nesta sección farase unha breve mención da estrutura da ferramenta (apartado 1.2.1) e da propia memoria (apartado 1.2.2) do proxecto.

1.2.1 Ferramenta

O resultado deste traballo non é un produto software final, senón a implementación de diversos algoritmos e unha serie de conclusións, resultado dun proceso de estudo, aprendizaxe e investigación. Todos os ficheiros que constitúen o código fonte das distintas versións dos algoritmos serán xestionados mediante un repositorio git, onde poder levar o control de versións do propio proxecto.

1.2.2 Memoria

Esta memoria está estruturada en seis capítulos, sendo o actual o primeiro de todos eles. A continuación, farase unha breve descrición dos restantes:

Capítulo 2: Fundamentos teóricos e tecnolóxicos

Este capítulo realiza unha introdución á temática do traballo. Aquí se expoñen os fundamentos teóricos e matemáticos que se utilizan no proxecto e se mencionan as ferramentas ou estudos xa existentes no ámbito do traballo, ademais de aquelas tecnoloxías que foron empregadas na realización do mesmo, xustificando a súa escolla.

Capítulo 3: Metodoloxía, planificación e custos

Neste capítulo comentarase o modo de traballo seguido durante a realización deste proxecto e a súa adaptación a un escenario de investigación.

Capítulo 4: Implementación

Este capítulo fai unha explicación, en orde cronolóxica, das distintas versións dos algoritmos desenvoltos. Tamén enumera os aspectos máis importantes dos mesmos e os problemas que motivaron a súa evolución.

Capítulo 5: Probas e resultados

Neste capítulo realízase a análise do rendemento dos algoritmos resultantes, confrontándoo co de solucións existentes. Para iso, empregaranse as ferramentas que proporciona o propio *framework* e os resultados utilizaranse para dirixir a liña de traballo.

Capítulo 6: Conclusións

Este último capítulo expón o resultado xeral, o resumo dos obxectivos acadados, posibles melloras a realizar sobre os algoritmos e futuras liñas de traballo.

Capítulo 2

Fundamentos teóricos e tecnolóxicos

Neste proxecto manéxanse conceptos e tecnoloxías que requiren dunha breve explicación previa para poder comprender o seu papel no traballo. Polo tanto, este capítulo serve de introdución á temática do traballo e expón os fundamentos teóricos e matemáticos que se utilizan no mesmo. Comezaremos na sección 2.1, explicando que é a operación de convolución, as distintas formas de abordala e cales son as súas aplicacións, en especial aquelas relacionadas coas redes neuronais. A continuación, na sección 2.2, mencionaremos as tecnoloxías empregadas para a realización do proxecto e, finalmente, a sección 2.3 servirá para facer unha breve mención daquelas tecnoloxías que xa implementan este tipo de operación na súa versión por lotes.

2.1 Conceptos teóricos

Nesta sección comezaremos explicando a operación de convolución dende o seu punto de vista matemático (apartado 2.1.1). No apartado 2.1.2 razoaremos por que ten tanta importancia nas redes de aprendizaxe profunda, entre as que destacan as redes neuronais convolucionais, e repasaremos distintos tipos de redes. Remataremos a sección comentando varios algoritmos que podemos empregar na súa implementación (apartado 2.1.3).

2.1.1 A operación de convolución

A convolución é unha operación matemática que se define como a integral do produto entre dúas funcións, f e g, tras desprazar unha delas unha distancia t. O resultado é outra función que representa como lle afecta a forma dunha á outra.

$$(f * g)(t) \doteq \int_{-\infty}^{\infty} f(\eta)g(t-\eta)d\eta \tag{2.1}$$

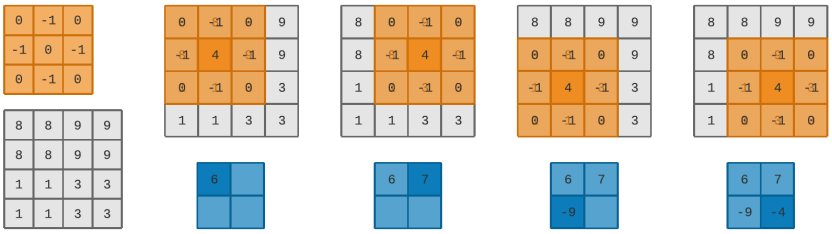


Figura 2.1: Operación de convolución en dúas dimensións sobre unha imaxe (en gris) de 4x4 píxeles, e empregando un filtro (en laranxa) de 3x3 píxeles.

Se f e g son funcións discretas podemos reescribir a fórmula substituindo a integral por un sumatorio:

$$f[m] * g[m] = \sum_{n} f[n]g[m-n] \tag{2.2}$$

Agora ben, esta operación pode trasladarse ao procesamento de imaxes empregando funcións de varias dimensións. Neste caso o que desprazaríamos sería unha imaxe (filtro) sobre outra, e avanzaríamos unha unidade (píxel) de cada vez. Observemos a figura 2.1, neste exemplo realizamos a operación cun filtro de 3x3 píxeles sobre unha imaxe de 4x4, ambas son imaxes de dúas dimensións. O resultado será unha nova imaxe de unicamente 2x2 posicións, cuxos valores serán a suma do resultado de multiplicar o valor de cada píxel do filtro polo valor do píxel da imaxe sobre o que se atopa. É dicir, o primeiro píxel calcularíase da seguinte forma:

$$0 \cdot 8 + (-1) \cdot 8 + 0 \cdot 9 + (-1) \cdot 8 + 4 \cdot 8 + (-1) \cdot 9 + 0 \cdot 1 + (-1) \cdot 1 + 0 \cdot 3 = 6$$

Como podemos ver, a imaxe resultante é máis pequena cá de entrada. Se nos interesa conservar as dimensións orixinais podemos engadir un marco (*padding*) de píxeles ao redor da imaxe, o seu tamaño depende das dimensións do filtro. É moi habitual que todos os valores dese marco sexan 0 (*zero-pad*), pero tamén podemos replicar os valores lindeiros.

Agora sabemos en que consiste esta operación e podemos aplicala a imaxes en branco e negro (2D), pero se engadimos máis dimensións ao filtro e repetimos a operación podemos estendela a imaxes en cor (3D), nas que a información sobre a color engádese nunha dimensión a maiores, e incluso a vídeo (4D).

Esta operación pode sufrir variacións en función de se temos en conta outros parámetros como o *stride*, o número de unidades que avanzamos en cada iteración; ou a *dilation*, a separación entre os valores do filtro. [1]

2.1.2 Redes de aprendizaxe profunda

As redes de aprendizaxe profunda (DNN, do inglés *Deep Neural Network*) [2] son un tipo de rede neuronal artificial (ANN, do inglés *Artificial Neural Network*), un modelo computacional formado por neuronas, cada unha cos seus pesos e os seus *bias*, que se diferencian por ter moitísimas capas, de aí o seu nome.

As principais vantaxes deste tipo de redes son, por un lado, que ademais de problemas lineares (figura 2.2) poden resolver relacións non lineares moito máis complexas; e polo outro lado, que actúan sobre a totalidade dos datos, sen necesidade de acción humana no proceso de extracción de características. Esta é unha funcionalidade moi importante, pensemos por exemplo nun problema de clasificación de imaxes: extraer manualmente as características de cada imaxe procesada require de persoas cun gran coñecemento no dominio, e consume demasiado tempo. As DNN poden ser adestradas para automatizar todo ese traballo. Hai que ter coidado, non obstante, pois un dos retos aos que nos enfrontamos ao empregar DNNs é o sobreadestramento, máis propenso a ocorrer neste tipo de redes polas numerosas capas de abstracción que engaden.

A continuación, comentaremos as principais variantes de redes de aprendizaxe profunda e as aplicacións que ten cada unha delas.

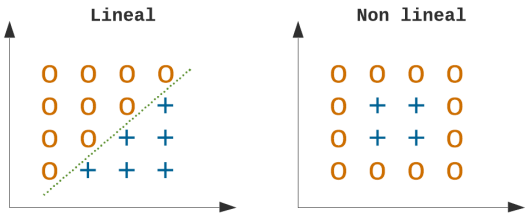


Figura 2.2: Exemplo dun problema de clasificación lineal e dun non lineal

CNN, *Convolutional Neural Networks*

A eficacia e utilidade das Redes Neuronais Convolucionais [3] deu visibilidade e recoñecemento ao *Deep Learning* de cara ao mundo. Debido a que a súa implementación asume que as entradas á rede son imaxes, son moi efectivas en tarefas de visión artificial, como a clasificación e a segmentación de imaxes, entre outras aplicacións. Unha visión global da súa arquitectura (figura 2.3), sen entrar en detalle, podería ser a seguinte:

1. Unha capa de entrada que se encarga de recibir os datos (imaxes), con tantas neuronas como elementos de procesamento (píxeles).

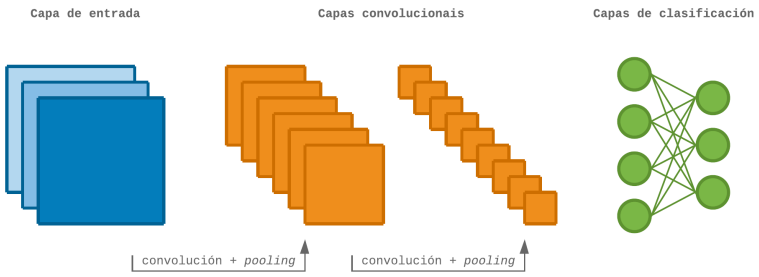


Figura 2.3: Arquitectura dunha Rede Neuronal Convolucional.

2. Un gran número de capas que realizan a extracción de características mediante dúas operacións:

   - convolución: lixeiramente distinta á convolución 2D que explicamos no apartado 2.1.1, verémola en detalle uns parágrafos máis adiante.
   - *pooling*: redución do tamaño dos datos para simplificar o procesado posterior e evitar o sobreadestramento.

   A profundidade da imaxe (o número de mapas de características) vai aumentando conforme esta avanza a través das sucesivas capas convolucionais, mentres que as súas dimensións (altura e largura) diminúen polo proceso de *pooling*. Os datos engadidos son información extraída das propias imaxes que serve de entrada á ultima fase, a de clasificación.

3. Unha última fase de clasificación, composta por múltiples capas totalmente conectadas. As características extraídas na fase previa permiten realizar unha análise dos datos moito máis detallada, sobre todo, destaca a capacidade de procesar as imaxes en anacos, podendo identificar obxectos independentemente da súa posición ou rotación.

A continuación, explicarase con maior detalle a particular implementación que fan estas redes da súa operación principal, a convolución.

Partindo dunha imaxe e un filtro cun número igual de canles, en primeiro lugar, realizan a convolución 2D entre as canles da imaxe e do filtro, o que produce unha nova imaxe co mesmo número de canles. A continuación suman entre si, píxel a píxel, os resultados de cada canle para eliminar a terceira dimensión da imaxe. O resultado é unha matriz en dúas dimensións que equivale a unha única canle da imaxe resultante. En caso de dispor de varios filtros, a repetición desta operación con cada un deles daría lugar a unha canle de saída distinta. Xuntas, esas matrices formarían o resultado da operación, unha nova imaxe en tres dimensión. A figura 2.4 pretende aclarar este proceso mediante un exemplo no que se emprega un único

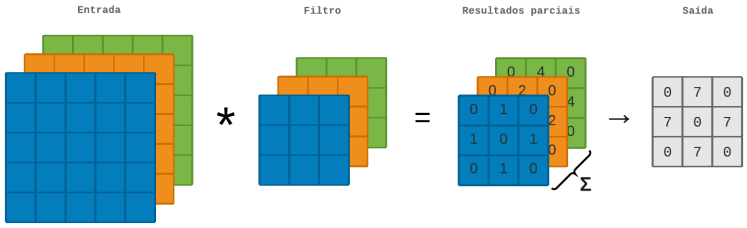


Figura 2.4: Convolución realizada por unha rede neuronal convolucional

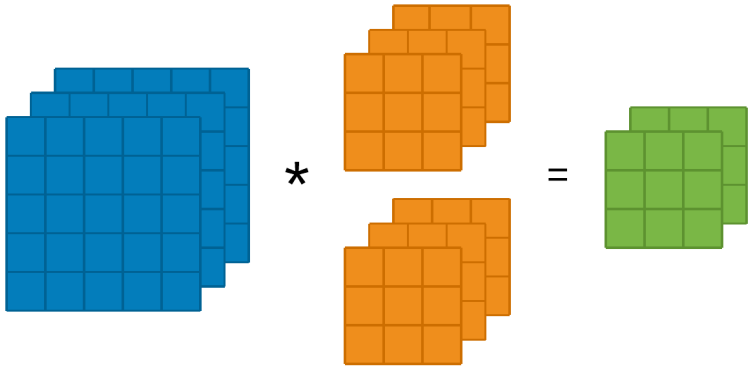


Figura 2.5: Relación entre o número de filtros e o número de canles da saída.

filtro. Se nos fixamos na figura 2.5 podemos ver que ao realizar esta mesma operación con dous filtros o resultado son dúas canles de saída distintas.

Neste punto imos engadir unha nova variable: o lote ou *batch*, que equivale ao número de imaxes a procesar. Como podemos ver na figura 2.6, se realizamos a operación descrita no anterior parágrafo sobre distintas imaxes e os mesmos filtros, obteremos dúas imaxes de saída, cada unha con tantas canles coma filtros foran utilizados. Polo tanto, e como temos que realizar a mesma operación sobre distintos datos, podemos aproveitar as capacidades dun dispositivo que dispoña de certa vantaxe neste tipo de operacións, por exemplo unha GPU, e buscar unha mellora do rendemento mediante a paralelización do procesamento.

Tendo en conta as equivalencias recollidas na táboa 2.1, podemos expresar esta operación mediante a seguinte fórmula:

$$y_{n,k,p,q} = \sum_{c}^{C} \sum_{r}^{R} \sum_{s}^{S} x_{n,c,p+r,q+s} \cdot w_{k,c,r,s} \tag{2.3}$$

Para todos os valores de n no conxunto [0, N-1] e todos os valores de k no conxunto [0, K-1].

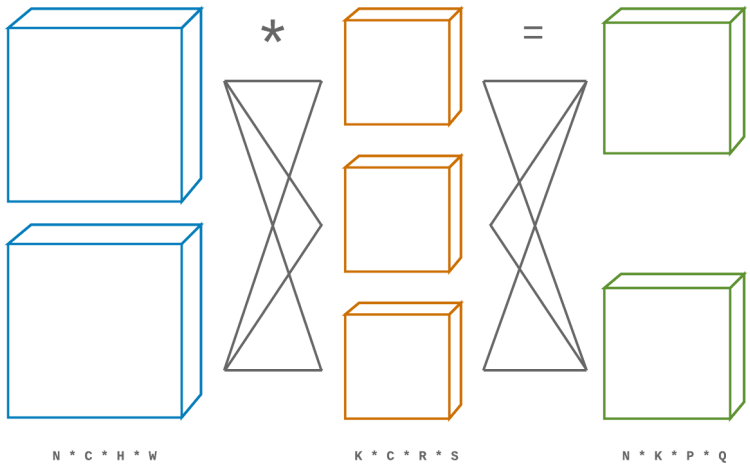


Figura 2.6: Relación entre o número de imaxes de entrada e o número de imaxes de saída.

Termo	Descrición
N	Número de imaxes no lote
K	Número de filtros (e canles de saída)
C	Número de canles da imaxe e o filtro
H	Alto, número de filas da imaxe
W	Largo, número de columnas da imaxe
R	Alto, número de filas do filtro
S	Largo, número de columnas do filtro
P	Alto, número de filas da saída
Q	Largo, número de columnas da saída

Táboa 2.1: Definición de variables comunmente empregadas no traballo

Xa para rematar, daremos un repaso ás aplicacións prácticas deste tipo de redes de aprendizaxe profunda.

A convolución é capaz de construír un espazo de características máis robusto, que proporciona moita información sobre o sinal de entrada. Debido á capacidade que teñen as CNN para aprender ditas características, son especialmente eficaces no recoñecemento de obxectos en imaxes. De feito, adoitan ser as mellores nos problemas de clasificación de imaxes. Poden identificar faces, individuos, sinais da rúa, e moitos outros aspectos dos datos visuais. Destacan na análise de texto mediante o recoñecemento de caracteres ou palabras e incluso son boas analizando son. Outro dos campos no que son de utilidade é a visión por computador, ou visión artificial, que ten aplicacións directas na tecnoloxía de coches autónomos, drones e aplicacións de accesibilidade para persoas invidentes. Unha vez estudado o seu funcionamento e coñecidas as súas aplicacións, podemos apreciar a gran utilidade desta operación no campo da intelixencia artificial.

RNN, *Recurrent Neural Networks*

As Redes Neuronais Recorrentes son un tipo de rede neuronal prealimentada [4] que integra retroalimentación, é dicir, os datos de saída dunha capa son utilizados como entrada da mesma na seguinte iteración. Desta forma permiten que a información persista durante un tempo, coma se dun estado se tratase. Esta característica permítelles procesar a información en xanelas temporais, o que as converte nas redes máis indicadas cando necesitamos modelar funcións nas que a entrada está composta por vectores cunha dependencia temporal entre os seus valores. Algúns problemas típicos que cumpren esta característica son o subtitulado de imaxes, a síntese de voz, o procesamento da música e da linguaxe, etc.

UPN, *Unsupervised Pretrained Networks*

A diferencia do adestramento supervisado que adoita utilizar datos etiquetados por persoas, as redes non supervisadas buscan patróns nun conxunto de datos sen etiquetas preexistentes e cun mínimo de axuda humana. Imos diferenciar tres tipos de redes non supervisadas:

- **Autoencoders**. O seu obxectivo é aprender a comprimir e codificar datos de forma eficiente para despois poder reconstruílos a partir da súa representación codificada, intentando que se asemelle todo o posible á entrada orixinal. Como moitas outras redes, poden ser parte dunha rede máis grande ou ser usadas de forma individual. As súas aplicacións son problemas de detección de anomalías, eliminación de ruído ou aqueles nos que se necesita reducir a dimensionalidade dos datos, normalmente como unha fase previa que simplifica o posterior procesado. [5]

- **DBN, *Deep Belief Networks***: A arquitectura destas redes ten dúas fases, unha formada por varias capas de máquinas de Boltzmann (RBM, *Restricted Boltzmann Machines*) [6] sobre as que se realiza o preadestramento sen supervisión, e outra consistente nunha rede neuronal prealimentada [4] típica, cun adestramento supervisado para refinar os resultados. As súas aplicacións se centran no procesamento de imaxes.

- **GAN, *Generative Adversarial Networks***: Este tipo de redes se caracterizan por adestrar dous modelos ao mesmo tempo. Un deles, o modelo "xenerativo", sintetiza datos novos a partir dun gran conxunto de adestramento. O outro modelo, o "discriminante", consume eses datos coma entradas e intenta clasificalos como reais ou sintéticos. O obxectivo é adestrar á primeira rede para que constrúa uns datos tan verosímiles que consigan enganar á segunda.

  Son aplicables en todo tipo de problemas nos que haxa que sintetizar datos: imaxes a partir de descricións, imaxes a partir de outras imaxes, voz, son, vídeo, etc.

En resumo, dependendo das aplicacións, elixiremos un tipo de rede ou outro. Para xerar datos (imaxes, audio, texto, etc.) usaremos GANs, *autoencoders* ou Redes Neuronais Recorrentes; para procesamento de imaxes, CNNs ou DBNs; e en caso de que teñamos que procesar datos secuenciais, usaremos tamén RNNs.

2.1.3 Algoritmos

Existen distintas formas de abordar a implementación da convolución. Podemos, por exemplo, crear un algoritmo a partir da súa definición, é dicir, que implemente a fórmula que definimos na sección anterior (ecuación 2.3). Mais este algoritmo, que poderíamos chamar "normal", ten demasiada complexidade computacional polo que o seu uso resulta inviable en problemas relativamente grandes. Para solucionalo podemos empregar outros como GEMM, FFT ou Winograd [7], que reducen a complexidade da operación.

GEMM

É un dos algoritmos máis empregados. Está baseado na operación BLAS GEMM (*Basic Linear Algebra Subprograms*, *General Matrix Multiply*) e consiste en reordenar os datos de entrada de forma que, cunha multiplicación de matrices entre os datos reordenados e o filtro, obteñamos o resultado da convolución. O resultado da transformación é unha matriz cunha columna por cada posición recorrida ao deslizar o filtro sobre a matriz de entrada. O contido de cada columna son os píxeles cubertos polo filtro nesa posición e en todas as canles. Se antes tiñamos unha operación complexa, agora temos dúas máis sinxelas: a transformación, comunmente denominada *im2col*, e o produto de matrices. A parte positiva é que a multiplicación de matrices é unha operación que leva moito máis tempo sendo estudada, polo que

existen numerosas optimizacións sobre ela e incluso os propios dispositivos implementan mecanismos para acelerar esta operación.

A maior desvantaxe deste algoritmo é a cantidade de memoria que necesita para almacenar o resultado da transformación, que é significativamente maior que a matriz de entrada orixinal. Sendo as dimensións da imaxe de entrada *NxCxHxW*, as do filtro *KxCxRxS* e as da imaxe de saída *N*K*P*Q*, as dimensións da matriz transformada calcúlanse da seguinte forma:

$$H' = C \cdot R \cdot S$$
$$W' = P \cdot Q$$

Onde *H'* e *W'* son o alto e o largo da matriz transformada e o resto de parámetros poden ser consultados na táboa de equivalencias 2.1. A imaxe transformada, polo tanto, ten tantas filas coma píxeles un único filtro ($C \cdot R \cdot S$) e tantas columnas coma píxeles unha única canle da imaxe de saída ($P \cdot Q$). No capítulo 4 explicarase a implementación que se fixo deste algoritmo no proxecto, alí representaremos graficamente (figura 4.8) a operación de transformación levada a cabo na función *im2col*.

FFT

Os algoritmos baseados na Transformada de Fourier Rápida (FFT, do inglés, *Fast Fourier Transform*) son unha técnica moi coñecida e empregada no procesamento de imaxes. Consisten en trasladar o cálculo da convolución do dominio do tempo ao dominio da frecuencia, a través da Transformada de Fourier, onde podemos expresalo como un simple produto. Isto é posible grazas ao Teorema da Convolución [8], segundo o cal a Transformada de Fourier da convolución de dúas entradas é igual ao produto da Transformada de Fourier de cada unha desas entradas:

$$\mathcal{F}[f * g] = \mathcal{F}[f] \cdot \mathcal{F}[g] \tag{2.4}$$

Dado que a multiplicación é moito menos custosa que a operación de convolución, a vantaxe deste algoritmo dependerá do tempo de procesamento que necesitemos para realizar: dúas Transformadas de Fourier, unha a cada entrada, e unha Transformada de Fourier Inversa ao resultado do produto. Podemos observar mellor as operacións necesarias expresando a ecuación 2.4 deste outro xeito:

$$f * g = \mathcal{F}^{-1}[\, \mathcal{F}[f] \cdot \mathcal{F}[g] \,] \tag{2.5}$$

Onde $\mathcal{F}$ representa a Transformada de Fourier e $\mathcal{F}^{-1}$ representa a Transformada de Fourier Inversa. En xeral, o rendemento deste algoritmo aumenta co número de imaxes e filtros a pro-

cesar, xa que desta forma pode reutilizar moitas das transformadas que debe realizar en cada convolución.

Winograd

Esta implementación baséase nos algoritmos de filtrado mínimo de Shmuel Winograd [9], empregados no eido do procesamento de sinais pola súa utilidade na aplicación de filtros FIR (do inglés, *Finite Impulse Response*). O algoritmo consiste en realizar certas transformacións [10] sobre os datos de entrada para reducir ao mínimo o número total de multiplicacións que fan falla na operación. A cambio, aumenta o número de sumas que se precisan, mais estas adoitan ser computacionalmente menos custosas. Debido ás características do algoritmo, o número de sumas necesarias aumenta de forma cuadrática co tamaño das imaxes de entrada, por iso estas transformacións son máis axeitadas cando as entradas son máis ben pequenas.

2.2 Tecnoloxías: CUDA

CUDA foi a principal ferramenta empregada no proxecto. Por iso, no apartado 2.2.1 desta sección comezaremos facendo unha breve introdución á ferramenta, para despois, no apartado 2.2.3, explicar o modelo de programación desta linguaxe.

2.2.1 Introdución

NVIDIA CUDA® (*Compute Unified Device Architecture*) é unha plataforma de computación paralela que permite realizar computación de propósito xeral nunha GPU (GPGPU, do inglés: *general-purpose computing on graphics processing units*). Trátase dunha libraría coa que se pode interactuar dende C/C++, tamén se pode usar en Python, Fortran e Java a través de adaptadores.

Para contextualizar a escolla deste *framework*, por un lado é necesario comentar que NVIDIA é un dos principais fabricantes de tarxetas gráficas, xunto con AMD e Intel. Ademais, ten unha gran comunidade de desenvolvedores, probablemente debido a que as ferramentas que proporciona, malia ser en gran parte *software* propietario, están ben documentadas e contan cunha gran traxectoria. Podemos comprobalo botando unha ollada á cantidade de traballos académicos que utilizan a API de CUDA e todas as ferramentas que empregan este *framework* para optimizar as súas funcións.

Por outro lado, se dispoñemos dunha GPU dun modelo determinado, é lóxico que escollamos as ferramentas que o propio fabricante proporciona para a súa programación, xa que adoitan estar máis optimizadas. É o caso de CUDA e OpenCL. Á primeira en xeral proporciona

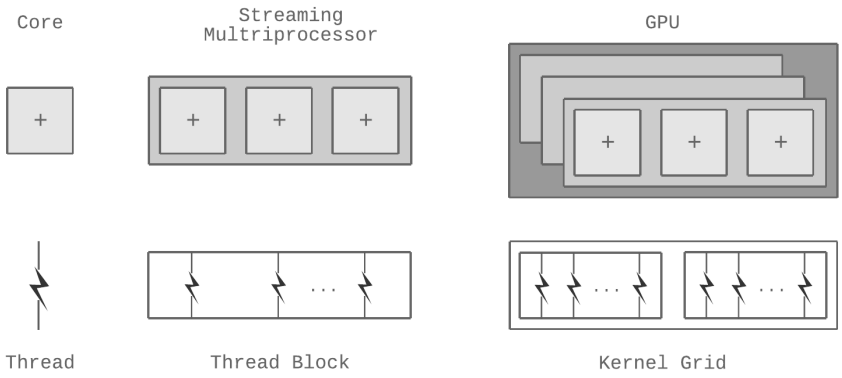


Figura 2.7: Correspondencia entre SM (*Streaming Multriprocessors*) e os elementos do *grid*

mellores resultados, mentres que o soporte de NVIDIA para OpenCL, aínda que dispoñible[1], é bastante deficiente. En definitiva, isto se traduce en que o mesmo código implementado en CUDA e OpenCL ten un rendemento bastante mellor en CUDA. Ademais, ao non ter coñecementos previos dunha ferramenta alternativa, a escolla dunha ou outra non supón ningún sobrecusto temporal. Estas son as razóns principais polas que se utilizou CUDA no desenvolvemento deste traballo.

2.2.2 A nivel hardware

Para ter unha idea completa do funcionamento da GPU, e entender mellor a eficacia de certas melloras que introduce o código CUDA, cómpre coñecer como se estrutura unha GPU de NVIDIA a nivel *hardware*, as súas estruturas máis básicas e como se relacionan estas cos conceptos de fío, bloque, e *grid*, que son unha abstracción empregada dende a perspectiva do programador. En liñas xerais, agrúpanse os fíos que executan a mesma instrución en *warps*, de 32 fíos nas GPUs actuais. Varios *warps* constitúen un bloque de fíos. Varios bloques de fíos asígnanse a un SM (do inglés, *Streaming Multiprocessor*). Finalmente, os distintos SM constitúen a unidade da GPU, que executa toda la malla do *kernel*. Podemos velo graficamente na figura 2.7. A continuación, describiranse con máis detalle estes conceptos.

Streaming Multiprocessors (SMs)

As distintas arquitecturas das GPU de NVIDIA, como poden ser a Kepler[2] ou a Volta[3], constitúense en torno a un gran conxunto de *Streaming Multiprocessors*. Cando un programa

[1] Implementación de OpenCL para tarxetas gráficas de NVIDIA - https://developer.nvidia.com/opencl

[2] Kepler Architecture for High Performance Computing - https://www.nvidia.com/pt-br/data-center/nvidia-kepler/

[3] Artificial Intelligence Architecture: NVIDIA Volta - https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/

CUDA invoca un *kernel*, os bloques do *grid* enuméranse e distribúense entre os SM con capacidade de execución dispoñible. Estes non son máis que multiprocesadores de propósito xeral cunha baixa frecuencia de reloxo (polo tanto, enerxeticamente máis eficientes) e pequenas cachés. Para maximizar a utilización das súas unidades funcionais, implementan o paralelismo a nivel de fío, máis que o paralelismo a nivel de instrución dentro dun único fío. A diferencia dos núcleos dunha CPU, execútanse en orde e non hai predicción de salto nin execución especulativa. Un SM pode executar moitos fíos concorrentemente, e tamén varios bloques, ata un límite que varía en función da implementación. A medida que os bloques rematan a súa execución, vanse lanzando novos bloques nos multiprocesadores que quedan libres.

Warps

O multiprocesador crea, xestiona, programa e executa os fíos en grupos de 32 chamados *warps*. Cando un multiprocesador recibe un ou mais bloques de fíos para executar, divídeos en *warps* que son programados por un *warp scheduler* para a súa execución. A forma na que un bloque é dividido en *warps* é sempre a mesma: cada *warp* contén fíos de identificadores consecutivos e crecentes contendo o primeiro *warp* o fío número 0.

Un *warp* executa unha instrución de cada vez, polo que a eficiencia total é alcanzada cando os 32 fíos dun *warp* concordan no seu camiño de execución. Se os fíos dun mesmo *warp* diverxen a través dunha rama condicional que dependa dos datos, o *warp* executa en serie cada camiño tomado, desactivando os fíos que non están nese camiño, e cando todos os camiños están completos, os fíos converxen de volta no mesmo camiño de execución. A diverxencia de camiños ocorre apenas dentro dun *warp*, os diferentes *warps* execútanse independentemente de estaren a executar camiños de código comúns ou disxuntos.

Os multiprocesadores executan centos de fíos ao mesmo tempo. Para manexar tal cantidade de fíos, apóianse nunha arquitectura chamada SIMT (*Single-Instruction, Multiple-Thread*) [11] similar á SIMD (*Single-Instruction, Multiple-Data*) na medida en que unha única instrución controla varios elementos de procesamento. A diferencia chave é que as instrucións SIMT especifican a execución e o comportamento nos saltos dun único fio. A diferencia do SIMD, o SIMT permite aos programadores escribir código paralelo a nivel de fío para fíos independentes, escalares, así como código paralelo a nivel de datos para fíos coordinados.

2.2.3 Modelo de programación

Supoñamos que partimos dun programa secuencial escrito en C++. Tras unha análise do rendemento do código, a través de ferramentas de *profiling*, podemos detectar que a meirande parte do tempo de execución se gasta nunha pequena porción de código. Se isto supón un

problema para o rendemento do noso programa, podemos decidir implementar en CUDA o código problemático, transformándoo nun ou en varios *kernels* nos que paralelizar o seu procesamento. Ademais das modificacións no código, precisaremos cambiar tamén as instrucións de compilación, empregando o compilador `nvcc` que proporciona o propio *framework*. Desta forma poderemos executar funcións nunha GPU de NVIDIA.

Todo programa CUDA consta, como mínimo, dos seguintes pasos:

1. Reserva de memoria tanto no *host* (CPU) coma no dispositivo (GPU). E inicialización dos datos no *host*.
2. Copia dos datos do *host* ao dispositivo.
3. Execución paralela da función (kernel) na GPU.
4. Copia dos resultados do dispositivo ao *host*.
5. Liberación dos datos da memoria.

A continuación explicaranse con detalle algúns conceptos.

Kernels

As funcións que se executan na GPU chámanse *kernels*. Para definir un *kernel* basta con anotar unha función coa directiva `__global__`:

```
__global__
void add(int n, float *x, float *y) {
    // ...
}
```

Os *kernels* execútanse na GPU en bloques de fíos (*threads*). Á hora de executar o *kernel* temos que indicar, como mínimo, o número de bloques e o número de fíos por bloque que imos lanzar. Unha chamada ao *kernel* pode conter os seguintes parámetros:

```
add <<< gridDim, blockDim, nShared, stream >>>(N, x, y);
```

Onde:

- `gridDim` especifica o número de bloques do *grid*. Podemos indicalo cun número enteiro, pero se trata dunha variable de tipo `dim3` que permite lanzar *grids* multidimensionais.

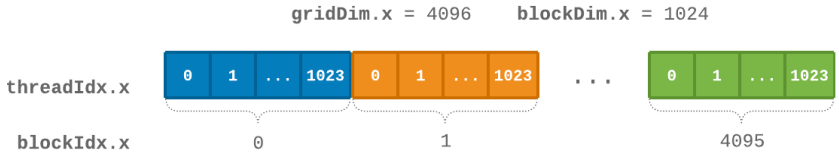


Figura 2.8: Esquema da distribución dos fíos en bloques no *grid* de CUDA

- `blockDim` especifica o número de fíos por bloque. Igual que o anterior, é unha variable de tipo `dim3`. Veremos como declarar variables deste tipo no seguinte apartado.
- `nShared` é de tipo `size_t` e especifica o número de bytes de *Shared Memory* que se teñen que reservar en cada bloque de forma dinámica. Trátase dun argumento opcional que por defecto é 0. Verémolo cando expliquemos a *Shared Memory*.
- `stream` é de tipo `cudaStream_t` e especifica o *Stream* [12] asociado a esta chamada ao *kernel*. Dito *Stream* debe estar aloxado no mesmo bloque no que se efectuou a chamada. Trátase dun argumento opcional que por defecto é 0.

Ademais, o código do *kernel* debe ser escrito de forma xenérica, de modo que, en base a uns identificadores de fío accesibles dende o código do *kernel*, dirixa a execución de cada fío e reparta o traballo que teñen que realizar, de forma individual se é necesario.

Por outro lado, como explicamos no apartado anterior, os bloques divídense en *warps* a nivel hardware, que son grupos de 32 fíos. Por iso, para acadar un alto valor de ocupación [13] da GPU, é recomendable elixir valores múltiplos de 32 no número de fíos por bloque.

Grid

CUDA é unha linguaxe esencialmente paralela, polo tanto deseñada para repartir o traballo entre os múltiples procesos que executan o *kernel*. Ditos procesos organízanse nunha malla ou *grid*, dividida en bloques de fíos (*threads*). Os límites ao número e tamaño dos bloques veñen determinados polo hardware da tarxeta onde se executa o código.

Na figura 2.8 vemos un exemplo de como se organizarían 4096 bloques de 1024 fíos cada un. Neste exemplo empregamos a seguinte nomenclatura: `gridDim.x`, número de bloques; `blockDim.x`, número de fíos en cada bloque; `blockIdx.x`, índice do bloque actual; e `threadIdx.x`, índice do fío actual dentro do bloque. A partir destes valores, poderíamos calcular o índice absoluto dun fío dentro do *grid*, da seguinte forma:

$$blockIdx.x * blockDim.x + threadIdx.x$$

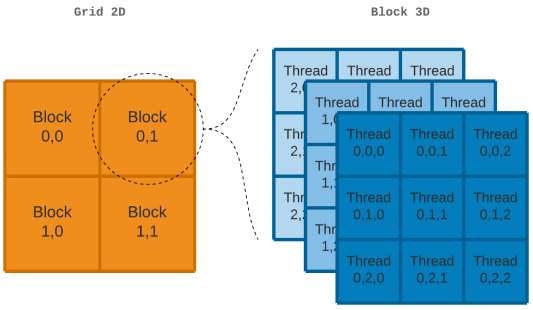


Figura 2.9: Exemplo de distribución dos fíos nun *grid* de dúas dimensións e bloques de tres dimensións.

Estes son os nomes dunhas variables accesibles dende calquera *kernel* CUDA, que identifican de forma unívoca ao fío que está a executar o *kernel*. As dimensións, tanto dos bloques coma do *grid* debemos indicalas no momento da chamada ao *kernel*, tal e como víamos no apartado anterior.

Non obstante, os exemplos que proporcionamos ata o momento tan só teñen en conta *grids* dunha única dimensión. A realidade é que podemos invocar os procesos de forma que se organicen en dúas e ata tres dimensións, a nivel de *grid* e a nivel de bloque. Para iso, temos que declarar dúas variables de tipo `dim3` coas dimensións do *grid* e dos bloques:

```
dim3 gridDim ( gx, gy, gz );
dim3 blockDim( bx, by, bz );
```

E utilizalas na chamada ao *kernel* que explicamos no apartado anterior. Tal e como víamos hai uns parágrafos para o caso dunha única dimensión, podemos acceder a estes valores dende o código do *kernel* a través dos atributos `x`, `y` e `z` das variables predefinidas `gridDim` e `blockDim`, e aos identificadores de fío de cada dimensión usando a mesma nomenclatura. Na figura 2.9 podemos ver como se organizarían catro bloques tridimensionais nun *grid* de dúas dimensións.

Esta funcionalidade adoita ser útil en aplicacións con procesamento de imaxes, xa que permite asignar cada dimensión do *grid* cunha dimensión da imaxe. Por exemplo, se gardamos dita imaxe nun *array* de dúas dimensións e nos interesa dividilo espacialmente para procesalo por zonas, podemos utilizar un *grid* e incluso un bloque de dúas dimensións, tal e como podemos ver na figura 2.10. Por suposto, debemos aproximar as dimensións do *grid* ás da propia imaxe, e aínda así é posible que non todos os fíos dun bloque, especialmente aqueles que se

Figura 2.10: Exemplo de distribución dun *grid* e bloques en dúas dimensións para o procesamento dunha imaxe.

encargan de procesar os límites da imaxe, poidan seguir o mesmo camiño de execución.

Device Memory

Sabendo como lanzar o *kernel* e identificar o fío que o está a executar, é o momento de poñer á GPU a operar cos datos. Para iso, antes debemos copiar eses datos dende a memoria do *host* (CPU) á memoria do dispositivo (GPU). En primeiro lugar, temos que reservar espazo na memoria do dispositivo coa función `cudaMalloc` e despois copiar os datos almacenados na memoria do *host* coa función `cudaMemcpy`:

```
cudaMalloc(deviceData, n * sizeof(float));
cudaMemcpy(deviceData, hostData, n * sizeof(float),
    cudaMemcpyHostToDevice);
```

A continuación faríamos á chamada ao *kernel*, unha vez lanzado e, mentres a GPU traballa e modifica os datos, a CPU pode seguir executando instrucións. Finalmente, cando a GPU remata a súa execución, volveríamos copiar o resultado coa función `cudaMemcpy`, desta vez indicando a dirección contraria:

```
cudaMemcpy(hostResult, deviceResult, n * sizeof(float),
    cudaMemcpyDeviceToHost);
```

Finalmente, cómpre saber que cando lanzamos un *kernel* estamos facendo unha chamada asíncrona, é dicir, a CPU segue a executar o código que vén inmediatamente despois da chamada á función. Mentres que, cando chamamos á función `cudaMemcpy`, estamos facendo unha chamada síncrona, a CPU agarda a que todos os *threads* lanzados na GPU rematen o que estaban a facer antes de continuar a súa execución.

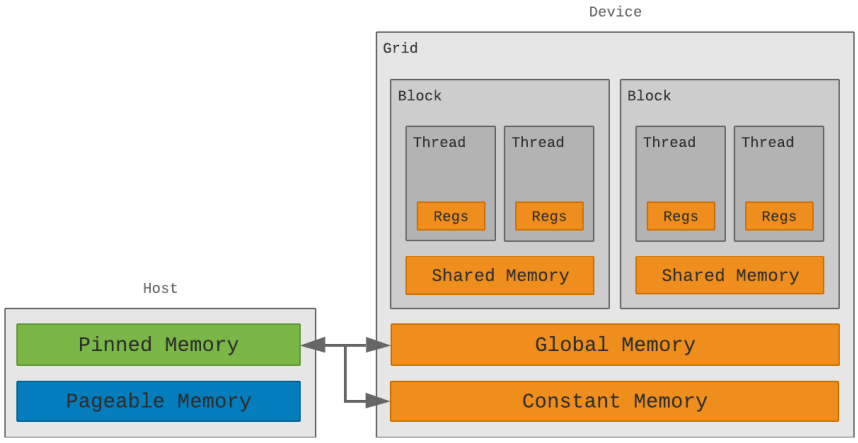


Figura 2.11: Tipos de memoria máis relevantes nunha GPU de NVIDIA

Mais a memoria do dispositivo é un concepto de alto nivel, en realidade está formado por distintos tipos de memoria. Na figura 2.11 podemos apreciar como se organizaría dentro do *grid* de CUDA a arquitectura de memoria dunha GPU de NVIDIA, no esquema están representados os tipos de memoria máis relevantes de cara a nosa función como programadores. A continuación imos dar un repaso a algúns dos tipos de memoria dispoñibles en CUDA, centrándonos naqueles que resultaron de máis utilidade neste proxecto.

En primeiro lugar temos a *Global Memory*, aquela que empregamos implicitamente ao copiar os datos dende a CPU; e ao mesmo nivel está a *Constant Memory*, como o seu nome indica, aquí podemos almacenar aquelas variables que permanecen constantes ao longo da execución do *kernel*. Estas dúas memorias son accesibles para todos os fíos (e para todos os bloques) que se están a executar, nas dúas os datos persisten ata a finalización da aplicación. En segundo lugar, a nivel de bloque, temos a *Shared Memory*, que explicaremos con máis detalle máis adiante. Para finalizar, cada multiprocesador dispón dun conxunto de rexistros de 32 bits para dividir entre os *warps*. Na táboa 2.2 temos un pequeno resumo coas diferentes formas de declarar unha variable en cada unha destas memorias dende un *kernel* CUDA.

Shared Memory

Este tipo de memoria, ao atoparse no mesmo chip, é escasa pero moitísimo máis rápida. A súa latencia teórica respecto á memoria global do dispositivo pode chegar a ser 100 veces menor. Debido a isto, é moi útil naqueles casos nos que necesitamos acceder repetidamente aos mesmos datos, pois aínda que é necesario facer unha transferencia de datos inicial dende a memoria do dispositivo, o que supón un pequeno sobrecusto, as perdas recupéranse rapidamente ao facer numerosas lecturas. Ademais, esta memoria é compartida a nivel de bloque,

Declaración	Memoria	Ámbito
`int variable;`	Register	Fío
`__shared__ __device__ int variable;`	Shared	Bloque
`__device__ int variable;`	Global	Grid
`__constant__ __device__ int variable;`	Constant	Grid

Táboa 2.2: Declaración de variables nos distintos tipos de memoria. Cando vai acompañado de `__shared__` ou `__constant__`, o modificador `__device__` é opcional.

é dicir, todos os fíos dun mesmo bloque teñen acceso á mesma *Shared Memory*. Isto permite organizar os fíos para que cada un cargue unha pequena parte dos datos que precisan na memoria compartida, e desta forma aforrar o maior tempo posible na primeira copia.

Existen dúas formas de reservar memoria compartida: a estática e a dinámica. Ámbalas dúas requiren do modificador `__shared__` diante. A estática é como instanciar un *array* de datos, precisamos saber o tamaño que vai ter á hora da compilación. Mentres que coa dinámica podemos indicar o tamaño no momento da execución, a través dun terceiro parámetro na chamada ao *kernel*:

```
1 dynamicExample <<< 1, n, n*sizeof(int) >>>(d_d, n);
```

Neste exemplo o tamaño reservado na memoria compartida sería `n*sizeof(int)`, é dicir, caberían `n` números enteiros. Na seguinte figura (2.12) podemos ver como teríamos que declarar as variables compartidas dependendo da súa asignación.

```
1 __shared__ int s[64];
```

(a) Estática

```
1 extern __shared__ int s[];
```

(b) Dinámica

Figura 2.12: Dúas formas de reservar memoria compartida

Global Memory

A memoria global forma parte da DRAM (*Device Random Access Memory*), á que accedemos a través de transaccións de 32, 64, ou 128 bytes. Para mellorar o rendemento global da memoria, cómpre ter en conta un concepto moi importante: a coalescencia (en inglés, *coalescing*) [14]. Esta capacidade vai afectar en gran medida ao rendemento dos nosos algoritmos.

Falamos de accesos a memoria coalescentes cando poden ser realizados conxuntamente nunha única transacción. Nas GPU de NVIDIA isto ocorre cando os 32 fíos que conforman un *warp* executan ao mesmo tempo unha instrución de carga (*load*) de datos consecutivos e aliñados. Para maximizar o rendemento dos accesos a memoria, é importante maximizar a coalescencia procurando:

- Seguir os patróns de acceso máis optimizados en función da *Compute Capability* do dispositivo.
- Usar tipos de datos que cumpran os requerimentos de tamaño e aliñamento. Os tamaños permitidos son 1, 2, 4, 8, e 16 bytes.
- Completar os datos cando non cumpran co tamaño mínimo nalgúns casos.

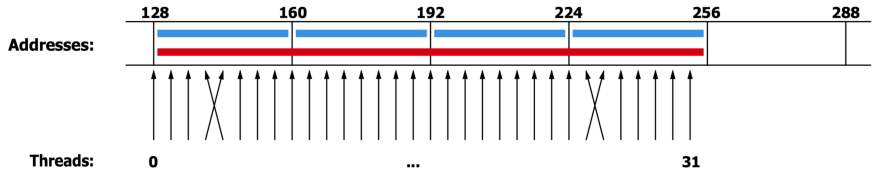
Supoñamos, por exemplo, que os 32 fíos que conforman un *warp* realizan unha operación de lectura sobre 128 bytes consecutivos, é dicir, 32 palabras de 4 bytes. Para que os 32 accesos poidan ser coalescentes e realizarse nunha única transacción, a parte de ser consecutivos en memoria, deben estar aliñados. Isto quere dicir que o enderezo de partida dos datos debe ser un múltiplo do tamaño do tipo de dato, e tamén o número de bytes lidos. Na figura 2.13[4] está representado graficamente o fenómeno do aliñamento. Na mesma figura podemos ver que, aínda que os enderezos de memoria son consecutivas, os accesos a memoria non son secuenciais (crúzanse as frechas). Isto non é un problema nos dispositivos de *Compute Capability* 3.0 e superior, onde os accesos seguen sendo coalescentes e se realizan nunha única transacción.

En resumo, para obter accesos coalescentes temos que empregar tipos de datos de 1, 2, 4, 8, ou 16 bytes e, ademais, o enderezo de partida dos datos debe ser un múltiplo dese tamaño. No caso dos *structs* de definición propia, podemos forzar o seu aliñamento utilizando os modificadores `__align__(8)` e `__align__(16)`:

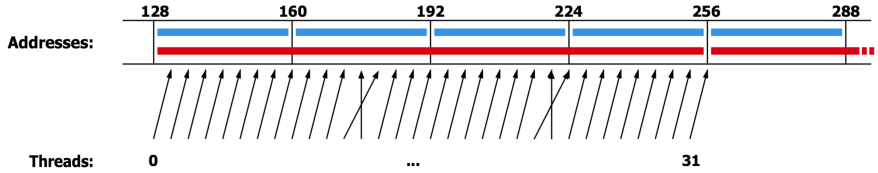
```
struct __align__(16) {
    float x;
    float y;
    float z;
};
```

Como se aprecia no exemplo anterior, o tamaño dos datos do *struct* non ten por que sumar o tamaño indicado no modificador (no caso anterior son $3 \cdot 4 = 12\ bytes$), o compilador completa os bytes restantes para cumprir co aliñamento.

[4]Imaxe inspirada nos esquemas da CUDA C/C++ Programming Guide - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capability-3-0



(a) Acceso a memoria aliñado



(b) Acceso a memoria desaliñado

Figura 2.13: Accesos que cumpren e non cumpren co aliñamento para realizarse nunha única transacción.

Pinned Memory

Neste punto podemos imaxinar que o feito de ter que copiar os datos da memoria do *host* á memoria do dispositivo vai supoñer un sobrecusto a maiores do traballo que desexábamos realizar. Existen distintas estratexias para reducir o tempo que ocupa esta transferencia e unha moi sinxela é colocar os datos directamente na *Pinned Memory*. Este tipo de memoria, aínda que non forma parte da memoria do dispositivo senón que se trata dun tipo de memoria protexida situada no *host*, podemos xestionala a través de funcións CUDA. Para entender mellor en que consiste a mellora que introduce é preciso coñecer como funciona o proceso de copia dos datos dende a CPU á memoria do dispositivo.

Cando reservamos memoria dende a CPU, por exemplo coa función *malloc* dende a linguaxe C, o espazo de memoria asignado aos datos por defecto admite paxinación. Isto significa que a CPU pode gardar eses datos fóra da memoria física, por exemplo no espazo de intercambio do disco, e dividilos en páxinas accesibles mediante a memoria virtual. Este tipo de memoria non está protexida, polo que os datos poden ser substituídos por outros de procesos que precisen máis memoria, nese caso, se alguén intenta acceder aos datos, pode producirse un fallo de páxina. A GPU non pode acceder directamente á memoria paxinada, polo que á hora de copiar os datos, ten que agardar a que o *driver* de CUDA reserve espazo na memoria protexida (*locked* ou *pinned*) e copie todas as páxinas necesarias a ese espazo temporal. Finalmente, a GPU pode volver copiar os datos á memoria do dispositivo, tal e como podemos ver na figura 2.14.

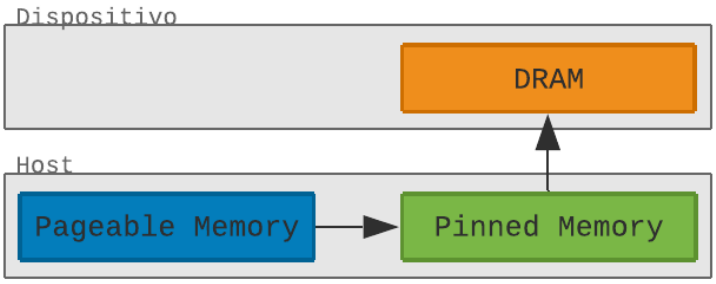


Figura 2.14: Tipos de memoria involucrados nunha transferencia de datos do *host* ao dispositivo

Como podemos ver, neste caso a *Pinned Memory* utilízase como un *buffer* temporal ao que a GPU pode acceder porque se corresponde directamente coa memoria física. Polo tanto, se dende un principio gardamos os datos na memoria protexida, a GPU poderá acceder aos datos sen axuda da CPU. Para lograr isto basta con substituir a función para reservar memoria (`malloc`) con `cudaMallocHost` ou `cudaHostAlloc`, e a función para liberar a memoria (`free`) con `cudaFreeHost`.

```
1 float *pinned;
2 cudaMallocHost((void**)&pinned, n * sizeof(float));
3 cudaFreeHost(pinned);
```

2.3 Estudos e tecnoloxías alternativas

A continuación, farase unha pequena análise das alternativas, tanto a nivel académico como tecnolóxico, no ámbito no que se desenvolve este traballo.

En primeiro lugar, existen numerosos estudos no ámbito da computación paralela en GPUs, moitos motivados pola necesidade de acelerar os procedementos levados a cabo nas redes de aprendizaxe profunda. En especial, cabe mencionar unha liña de investigación que seguen dende hai varios anos no BSC, o Centro de Supercomputación de Barcelona, chamada *Application optimization for GPU acceleration*[5]. O proxecto dispón de varias publicacións que serviron de referencia a este traballo.

En segundo lugar, entre as ferramentas de desenvolvemento máis empregadas en GPGPU,

[5]Todas as publicacións no seguinte enderezo: https://www.bsc.es/research-development/research-areas/programming-models/application-optimization-gpu-acceleration

CUDA é un dos *frameworks* máis usados. Non obstante, existe unha alternativa moi coñecida e que xa introducimos no apartado anterior: OpenCL. *Open Computing Language* é un *framework* de computación paralela, que permite aumentar o rendemento do código aproveitando as capacidades das plataformas heteroxéneas. Igual que CUDA, proporciona unha linguaxe de programación baseada en C++, aínda que neste caso OpenCL só define unha interface, a implementación para cada dispositivo depende do fabricante. Como xa dixemos anteriormente, a implementación de NVIDIA para OpenCL é bastante deficiente, por iso neste traballo consideramos que a mellor opción era utilizar CUDA.

Finalmente, se nos centramos nas diferentes tecnoloxías que implementan ou optimizan a operación de convolución, atopamos varias librarías, e a maior parte teñen as redes neuronais convolucionais como a súa principal aplicación. A continuación descríbense algunhas destas ferramentas, diferenciando aquelas de máis alto nivel (apartado 2.3.1), que xa implementan o algoritmo da convolución, de aquelas que se centran en acelerar as operacións de álxebra lineal subxacentes (apartado 2.3.2).

2.3.1 Librarías que optimizan rutinas das redes de aprendizaxe profunda

Se o que buscamos é optimizar unha aplicación que empregue redes neuronais convolucionais pero evitando a implementación do algoritmo da convolución no seu máis baixo nivel, poderíamos decantarnos por algunha das seguintes opcións.

cuDNN

A libraría NVIDIA CUDA® Deep Neural Network (cuDNN) [15] proporciona algoritmos moi optimizados, mediante o uso de CUDA, e moi empregados en redes de aprendizaxe profunda (DNN, do inglés *Deep Neural Networks*). Está pensada para ocultar os detalles de máis baixo nivel, optimizando as rutinas que máis se empregan na implementación das redes de neuronas, pero deixando o resto da implementación das mesmas nas mans do programador.

Entre as funcións que implementa atópanse os algoritmos máis coñecidos e utilizados da convolución en dúas dimensións: o normal, o directo, GEMM, FFT, Winograd e algunha variante dos tres últimos. Todos os algoritmos que implementa están preparados para realizar o procesado por lotes.

MKL-DNN (oneDNN)

Como sabemos, NVIDIA non é o único fabricante de dispositivos gráficos, Intel ten incluso máis volume de mercado e, aínda que non é propietaria, existe unha libraría ideada por

un dos equipos de MKL que permite executar código nos seus procesadores gráficos. OneAPI Deep Neural Network Library (oneDNN) [16], inicialmente coñecida como Intel® Math Kernel Library for Deep Neural Networks (Intel® MKL-DNN), é unha libraría de código aberto optimizada para os procesadores e gráficas de arquitectura Intel (produtos das series *Intel Architecture Processors, Intel Processor Graphics* e *Intel Xe*). A pesar de nacer do proxecto MKL e compartir certas similitudes, é unha libraría completamente independente, pensada para cooperar con outras ferramentas de *Deep Learning* como Caffe [17] (optimizado tamén para a arquitectura Intel) ou TensorFlow [18].

2.3.2 Librarías que optimizan rutinas da álxebra lineal

Se o que buscamos é realizar a nosa propia implementación do algoritmo, mais axudándonos de rutinas básicas xa optimizadas da álxebra lineal, poden resultar de utilidade as seguintes librarías, elixindo unha ou outra en función do hardware obxectivo.

cuBLAS

A libraría NVIDIA CUDA® Basic Linear Algebra Subroutine (cuBLAS) [19], é unha implementación dos subprogramas de álxebra lineal máis comúns optimizados a través da interface de programación de CUDA para a súa execución nas GPUs de NVIDIA. Dispón de tres APIs de programación distintas:

- cuBLAS: O usuario que emprega esta interface debe declarar el mesmo os datos na memoria do dispositivo e copialos dende a memoria do *host*, tal e como faríamos en CUDA.
- cuBLASXt: Empregando esta interface, o proceso de copia dos datos do *host* ao dispositivo é realizado pola propia libraría, segundo os parámetros indicados polo usuario.
- cuBLASLt: Esta interface expón unha versión máis lixeira da libraría e especializada nas operacións GEMM (do inglés, *General Matrix Multiply*), como a empregada nun dos algoritmos da operación de convolución que vimos anteriormente.

MKL

Outro gran fabricante no mercado dos procesadores, Intel, tamén dispón da súa propia ferramenta. A libraría Intel® Math Kernel Library (Intel® MKL) [20], optimiza o procesamento de rutinas matemáticas, aumentando o rendemento xeral das aplicacións e procurando ofrecer unha interface de programación sinxela. O hardware soportado inclúe todas as familias de procesadores Intel, mais ningún procesador gráfico. Está dispoñible en C, C++ e Fortran

de forma nativa, e en Python a través de adaptadores.

Entre as numerosas operacións que implementa, no que se refire a este traballo destacan as relacionadas coa álxebra lineal (BLAS, do inglés, *Basic Linear Algebra Subprograms*) e dentro destas as rutinas GEMM (do inglés, *General Matrix Multiply*) por lotes[6]. Tamén poderían ser de utilidade aquelas relacionadas coa Transformada de Fourier[7], xa que dispón de versións multidimensionais da FFT (do inglés, *Fast Fourier Transform*), usadas tamén nun tipo de algoritmos da convolución.

2.3.3 Outras ferramentas

A máis alto nivel dispoñemos de moitas librarías que utilizan a convolución ben para procesamento de imaxes, como OpenCV, ben para aprendizaxe profunda, como Keras. Estas dúas tamén poden empregar a GPU para acelerar as súas rutinas.

Keras

Keras é unha libraría de código aberto escrita en Python que pode executarse sobre TensorFlow, mais tamén noutras plataformas. Está enfocada na creación de todo tipo de redes de aprendizaxe profundo. Permite acelerar o seu adestramento empregando unha GPU de NVIDIA, pero de forma transparente para o programador.

OpenCV

OpenCV é unha coñecida libraría de procesamento de imaxes destinada principalmente a visión artificial. Aínda que non implementa a convolución por lotes, si que está dispoñible a convolución en dúas dimensións, e permite a súa execución tanto na CPU como na GPU. Para o segundo caso debemos empregar o seu módulo *gpu*[8] que, á súa vez, emprega a NVIDIA CUDA® Runtime API, polo que só é compatible con tarxetas gráficas de NVIDIA. Por outro lado, o seu módulo *dnn* permite importar e utilizar redes neuronais almacenadas en *frameworks* como Caffe ou TensorFlow, que poderíamos empregar para crear redes neuronais convolucionais, por exemplo.

[6] MKL: Linear Algebra - https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library/linear-algebra.html

[7] MKL: Fast Fourier Transform (FFT) - https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library/fast-fourier-transforms.html

[8] Guía da API de OpenCV. Introdución ao módulo *gpu*: docs.opencv.org/2.4/modules/gpu/doc/introduction.html

Capítulo 3

Metodoloxía, planificación e custos

Co obxectivo de organizar o traballo de forma eficiente, é conveniente realizar un pequeno esforzo previo que consista en determinar as tarefas pendentes e unha estimación do tempo que deberían ocupar. Alén disto, sobre todo tratándose dun entorno cambiante no que se desenvolven os proxectos de investigación, cómpre apoiarse nunha metodoloxía de desenvolvemento axeitada.

A estrutura do capítulo está formada por catro seccións. En primeiro lugar, na sección 3.1, exponse a metodoloxía de desenvolvemento escollida para a realización do proxecto. A continuación, na sección 3.2, explícase a planificación das tarefas a realizar seguindo dita metodoloxía. Despois, na sección 3.3, realízase unha estimación dos custos do mesmo e, finalmente, a sección 3.4 serve para deixar constancia do seguimento do traballo realizado.

3.1 Metodoloxía

Atopámonos ante un proxecto de investigación, cun equipo de desenvolvemento conformado por unha única persoa e no que o produto final pode sufrir diversos cambios en función dos camiños que deba tomar o proxecto. Por este motivo decidimos empregar unha metodoloxía de desenvolvemento áxil, baseada na realización de *sprints*, ou iteracións periódicas, tralas que avaliar o traballo realizado e planificar o traballo futuro. Esta metodoloxía, con base no marco de traballo Scrum mais adaptada ao carácter individual do traballo, pon o foco nas reunións e nos *sprints*, que rondan as tres semanas de duración, e busca sempre a simplicidade, para evitar sobrecargas na planificación. O ciclo de vida desta metodoloxía está representado no esquema da figura 3.1.

Seguindo esta filosofía, comezaremos cunha versión funcional pero ineficiente do algoritmo e iremos traballando sobre ela ata acadar un bo rendemento. En cada iteración, comezare-

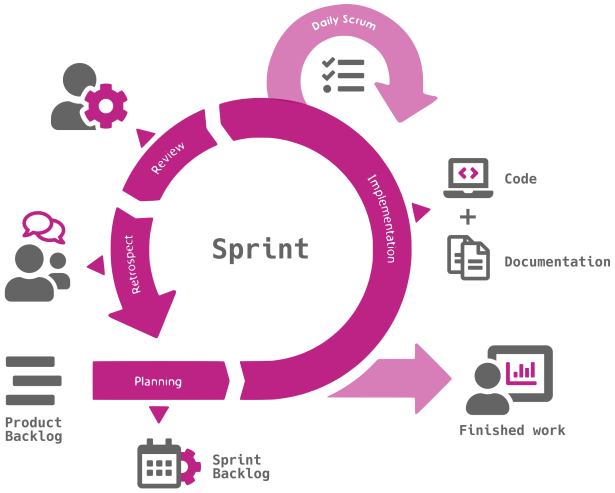


Figura 3.1: Ciclo de vida da metodoloxía adaptada ao proxecto

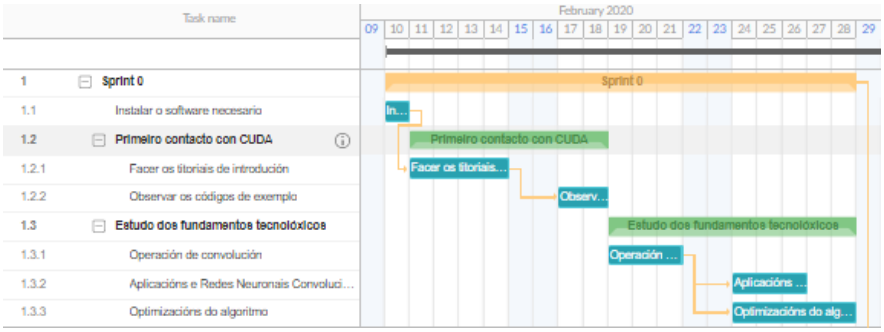
mos determinando e planificando as funcionalidades a desenvolver, para despois implementar ditas funcionalidades e documentar o traballo realizado. Para finalizar o ciclo, revisaranse os cambios realizados e comentaranse co titor nunha reunión que servirá, ademais, para esbozar o traballo da seguinte iteración.

Por outro lado, para organizar as tarefas de cada *sprint* e levar un rexistro do traballo realizado, empregouse unha ferramenta integrada en Microsoft Teams, Planner[1], da que dispoñemos coa nosa conta universitaria de Office 365. Esta ferramenta permite definir un taboleiro para cada iteración, onde introducir as tarefas que deben ser levadas a cabo en dito ciclo. As tarefas de cada taboleiro avanzan a través de varias listas en función do seu estado, por exemplo, neste traballo definíronse as seguintes listas: *To do*, para as que aínda non empezaron; *Doing*, as que están en curso; *Test/Blocked*, para as que requiren dalgunha acción para continuar e *Done*, para as finalizadas.

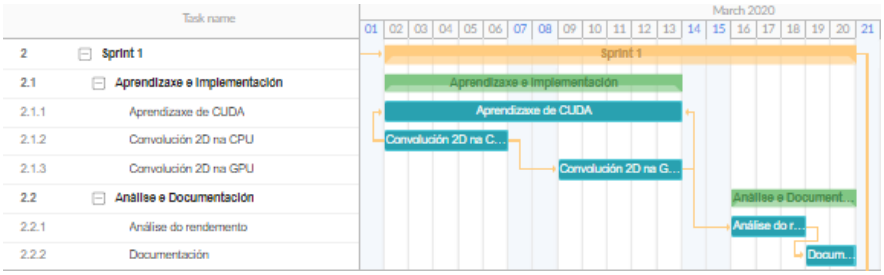
3.2 Planificación

Unha vez determinada a metodoloxía e seguindo as súas pautas, podemos comezar coa planificación inicial. Aínda que, dada a natureza do proxecto, non podemos determinar con exactitude o conxunto de tarefas que se desenvolverán durante a realización do mesmo, si podemos definir de forma xenérica unha serie de pasos que compartirán as distintas iteracións. Sabemos, polo tanto, que o proxecto estará dividido en *sprints* e cada un deles repetirá as seguintes fases:

[1]Microsoft Teams (https://teams.microsoft.com) e Microsoft Planner (https://tasks.office.com)



(a) Sprint 0



(b) Sprint 1

Figura 3.2: Diagrama de Gantt da planificación dos dous primeiros *sprints*

1. Aprendizaxe de CUDA e dos fundamentos matemáticos das operacións a realizar.
2. Proposta de variantes para a realización do procesado por lotes da convolución.
3. Deseño e implementación en CUDA de ditas variantes.
4. Análise e comparación do rendemento da proposta con outras implementacións.
5. Documentación do traballo realizado.

Desta forma, ao remate dun *sprint* obtemos un incremento ou mellora do traballo preexistente, e nos aseguramos de estudar, analizar e documentar o traballo realizado ata o momento.

O seguinte paso consiste en estimar a duración de cada iteración e ubicalas no calendario. Tomando como data de inicio do proxecto o 10 de Febreiro do 2020, podemos determinar que o tempo dispoñible é de 18 semanas en total. Polo tanto, podemos dividir o traballo en 6 *sprints*, de tres semanas cada un e 20 horas de traballo semanais, deixando tempo para dedicar ao resto dos estudos. En total

$$20\ horas/semana \cdot 3\ semanas/sprint \cdot 6\ sprints = 360\ horas$$

de duración do proxecto.

Recurso	Custo	Tempo	Total
Xefe de proxecto	30 €/h	30 h	900.0 €
Programadora	20 €/h	450 h	9000.0 €
Equipo gama media/baixa	16.6 €/mes	7 meses	116.2 €
Equipo gama media/alta	62.5 €/mes	7 meses	437.5 €
Software	-	-	0.0 €
Total			10453.7 €

Táboa 3.1: Avaliación dos custos do proxecto

Na figura 3.2 podemos ver a planificación inicial do proxecto, que comprendía os dous primeiros *sprints*: o chamado *Sprint 0*, que difire estruturalmente do resto por ser a toma de contacto coa materia, seguido do *Sprint 1* que xa dispón a estrutura que van ter o resto de *sprints*. Por suposto, conforme fomos avanzando na realización do proxecto, fomos planificando tamén o traballo a desenvolver. No Anexo A poden ser consultados os diagramas de Gantt correspondentes tamén ao resto de *sprints*.

3.3 Avaliación de custos

Aínda tratándose dun proxecto académico, imos facer unha pequena estimación dos custos económicos do traballo. Para abordar dita estimación, debemos ter en conta o gasto en persoal, materiais e licencias. Para empezar, podemos afirmar que os recursos necesarios deste proxecto son:

- Acceso a un equipo ou *cluster* cunha GPU de gama media/alta.
- Un PC cunha GPU de gama media/baixa no que levar a cabo o desenvolvemento.
- Librarías, compiladores e software necesario para desenvolver, compilar e executar código CUDA.
- Persoal: un xefe de proxecto (titor) e unha programadora (alumna).

Na táboa 3.1 é posible consultar a estimación realizada, onde as horas de traballo do titor foron obtidas da guía docente[2]. Ademais, aínda que non se dispoñía dos valores exactos, a amortización dos equipos informáticos calculouse supoñendo un custo inicial de 800€ e 3000€, dependendo da gama, e unha vida útil de 4 anos:

$$800\ € \ / \ (4\ anos \cdot 12\ meses/ano) = 16.6\ €/mes$$
$$3000\ € \ / \ (4\ anos \cdot 12\ meses/ano) = 62.5\ €/mes$$

[2]Guía Docente: Traballo Fin de Grao. Especialidade Enxeñaría de Computadores - https://guiadocente.udc.es/guia_docent/index.php?centre=614&assignatura=614G01099&any_academic=2019_20

Por outro lado, a duración do proxecto estendeuse máis aló do mes de Maio, acadando as 29 semanas dende o 10 de Febreiro ata o mes de Setembro. Calculamos, polo tanto, un total de 450 horas de traballo levado a cabo finalmente pola alumna, 90 horas a maiores das inicialmente planificadas e 150 horas se as comparamos coas indicadas na Guía Docente.

3.4 Seguimento

Nesta sección realízase unha pequena descrición do traballo realizado en cada iteración. O obxectivo é, por un lado, evidenciar o carácter iterativo do proxecto e, por outro, proporcionar unha visión xeral das diferencias e incrementos entre versións. A continuación, adicaremos un apartado a cada unha das iteracións.

3.4.1 Iteración 0

A primeira iteración centrouse na toma de contacto coa ferramenta (CUDA), na aprendizaxe ou repaso dos coñecementos necesarios da operación de convolución, o seu algoritmo e as posibles optimizacións sobre o mesmo. Tamén serviu para configurar e instalar o software necesario no meu computador persoal e na máquina do laboratorio.

3.4.2 Iteración 1

A continuación implementouse unha primeira versión da convolución 2D, que lía unha imaxe e un filtro desde ficheiro e escribía o resultado noutro. Esta versión aínda non consideraba o procesamento de varias imaxes.

3.4.3 Iteración 2

O seguinte paso foi implementar a convolución por lotes nunha segunda versión do programa. En primeiro lugar elaborouse, en linguaxe C++, un algoritmo secuencial executado por un único fío na CPU. Despois trasladouse dito algoritmo a un *kernel* CUDA, primeiro secuencial e finalmente paralelizado, co obxectivo de conseguir unha mellora do rendemento. Finalmente realizouse unha análise do rendemento da versión CUDA en comparación coa versión C++, así como a documentación dos resultados.

3.4.4 Iteración 3

Desta vez implementouse unha versión mellorada do algoritmo, chamada GEMM (do inglés, *General Matrix Multiply*), que divide a operación en dous: unha reordenación dos datos de entrada e unha multiplicación de matrices. Realizáronse dúas aproximacións distintas deste algoritmo.

3.4.5 Iteración 4

Na quinta iteración buscamos mellorar o algoritmo implementado previamente empregando novas estratexias de organización dos datos en memoria. Para iso empregamos ferramentas que proporciona CUDA como a *Shared Memory* ou a *Pinned Memory*.

3.4.6 Iteración 5

Por último, dedicamos a última iteración a rematar de redactar a presente memoria. Así como tamén foron realizadas algunhas probas adicionais que poñen de manifesto as características das diferentes versións do programa.

Capítulo 4

Implementación

Neste capítulo descríbese o proceso de implementación levado a cabo durante o proxecto. Explicaranse, en orde cronolóxica, aquelas versións do código que marcaron fitos de grande importancia no desenvolvemento do proxecto. Enumeraranse os aspectos máis importantes dos algoritmos desenvoltos en ditas versións e os problemas que motivaron a seguinte iteración, ou as posibles opcións de mellora. Todas as versións citadas nesta memoria atópanse etiquetadas no repositorio[1] co seu correspondente código de versión.

A estrutura do capítulo está dividida en seccións, unha por cada versión etiquetada no proxecto. Na primeira delas, a sección 4.1, explícase como foron os primeiros pasos na programación CUDA, implementando o algoritmo da convolución 2D. A segunda, a sección 4.2, detalla os cambios que foron necesarios para converter dito algoritmo na operación realizada polas capas convolucionais das redes de aprendizaxe profunda. Finalmente, na terceira e última sección (4.3), introdúcense varias optimizacións: un cambio de paradigma no algoritmo e dúas formas de acelerar os accesos á memoria.

4.1 Convolución 2D sen lotes (v1.0)

Esta primeira versión foi pensada para realizar unha iniciación á programación en CUDA e á súa libraría cuDNN. Nela realízase a convolución en dúas dimensións, implementando o algoritmo explicado no apartado 2.1.1 (figura 2.1), entre unha única imaxe e un único filtro. O código desta versión está sinalado coa etiqueta `v1.0` no repositorio[2].

As dúas entradas da operación, a imaxe e o filtro, son dúas matrices que se len ao principio do programa dende candanseus ficheiros. O resultado da operación, outra matriz das mesmas

[1] Repositorio de código fonte. Etiquetas - https://git.fic.udc.es/s.aguado/tfg/tags
[2] Etiqueta *v1.0* - https://git.fic.udc.es/s.aguado/tfg/tree/v1.0

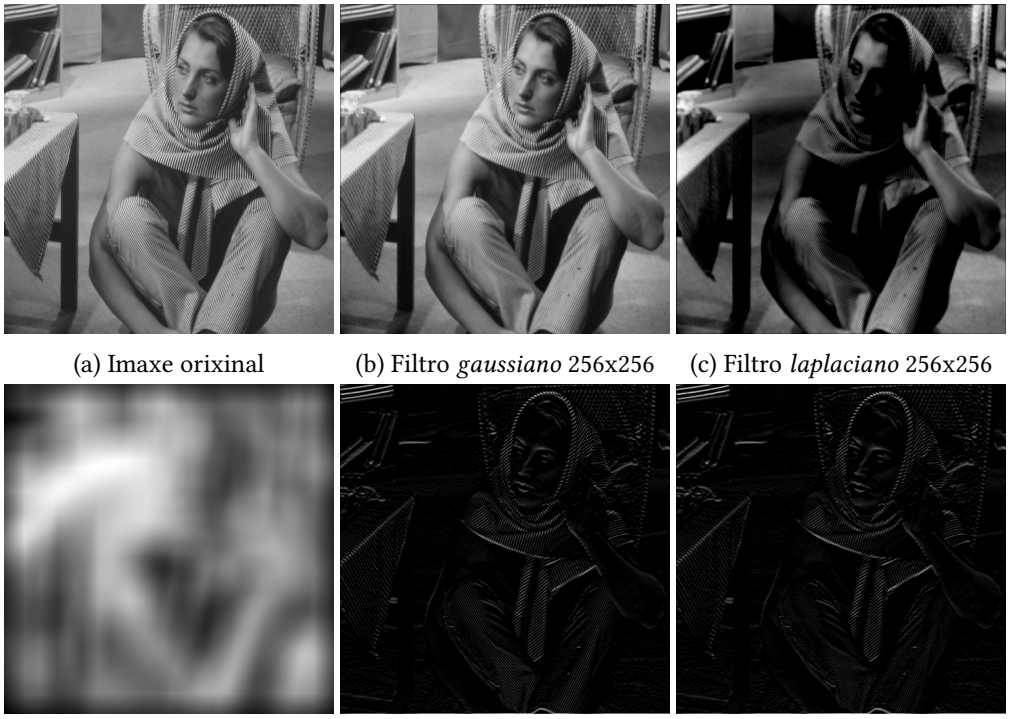
dimensións que a de entrada, escríbese como unha imaxe noutro ficheiro distinto. Isto permite visualizar os efectos dos distintos filtros sobre as imaxes. Para simplificar a lectura e escritura dos ficheiros das imaxes, que poden estar en múltiples formatos (*png*, *jpg*, *tif*, etc.), empregouse a libraría OpenCV. Con ela foron implementadas dúas funcións, `load_image` e `save_image`, para ler e escribir os ficheiros, respectivamente. Sen entrar nos detalles de implementación destas funcións, é interesante comentar que para evitar os erros por desbordamento durante o procesamento das imaxes, realizamos mediante chamadas a funcións de OpenCV unha normalización dos valores dos píxeles entre 0 e 1. Esta normalización consiste en escalar os valores dos píxeles de forma que se temos unha matriz de valores $[0\ 2\ 5\ 10]$ obteñamos outra equivalente de valores $[0.0\ 0.2\ 0.5\ 1.0]$. Polo tanto, a imaxe ten que ser convertida (se non estaba xa en dito formato) a unha matriz de números en punto flotante (*float*) de 32 bits. Ademais, para visualizar ben o resultado unha vez procesadas, a función de escritura converte todos os valores que resultaron negativos en ceros, para finalmente escribir o resultado nunha imaxe PNG.

Un detalle moi importante para despois codificar correctamente o algoritmo é o formato da disposición dos datos[3] das imaxes, neste caso, o que utiliza OpenCV. Este formato representa a disposición dos datos en memoria, verémolo con máis detalle ao comentar o programa cuDNN, onde xoga un papel moi relevante pola súa influencia no rendemento de certos algoritmos. De momento basta con saber que, nesta versión, tivemos que acceder ás posicións de memoria onde están almacenadas as matrices comezando polas súas dimensións (altura e largura, nesa orde) e finalizando co número de canles.

Unha vez lidas as imaxes é o momento de facer o mesmo cos filtros. Unha vez máis, definimos unha función para realizar esta tarefa: `read_filter`. A súa operación é tan simple como reservar memoria e gardar o contido do ficheiro como unha matriz de dúas dimensións. Mais, para operar con imaxes en cor, temos que engadirlle unha terceira dimensión. Para lograr isto, a función `load_filter` replica a matriz inicial (plantilla) tantas veces como canles teña a imaxe. Existen moitos tipos de filtros [21], segundo os valores que teñan realizarán distintas operacións sobre as imaxes. Algúns dos que podemos atopar no repositorio son os seguintes:

- `gaussian`: Os filtros *gaussianos* suavizan as imaxes e eliminan o posible ruído que poidan ter, canto máis grandes son, máis píxeles da imaxe orixinal ocupan e máis visible é o seu efecto.

[3] cuDNN Developer Guide: Data Layout Formats - https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html#data-layout-formats



(a) Imaxe orixinal (b) Filtro *gaussiano* 256x256 (c) Filtro *laplaciano* 256x256

(d) Filtro de media 64x64 (e) Filtro Sobel 3x3 (f) Filtro Prewitt 3x3

Figura 4.1: Resultados da execución dos programas *conv.cu* e *conv.cpp* sobre unha imaxe de 512x512 píxeles e distintos filtros.

- `average`: De forma similar, os filtros de media desenfocan as imaxes, máis non conseguen eliminar o ruído, o que si fan os filtros de mediana.
- `log` / `laplacian`: Os filtros *laplacianos*, tamén chamados LOG (do inglés *Laplacian Of Gaussian*), destacan as zonas nas que a intensidade dos píxeles cambia de forma brusca, por iso son moi empregados na detección de bordes nos obxectos.
- `prewitt` e `sobel`: Igual que o *laplaciano* serven para detectar bordes. Mais a diferenza deste, para conseguir o mesmo efecto fan falla dúas convolucións separadas, unha para detectar os bordes verticais e outra para detectar os horizontais. O resultado das convolucións súmase máis tarde. A diferenza entre ambos radica en que o filtro Sobel pon máis énfase nos valores do centro do filtro.

Na figura 4.1 móstrase o resultado de executar os programas empregando unha imaxe de 512x512 píxeles e algúns destes filtros. Debido ás considerables dimensións dos filtros empregados nas probas desta versión, todos eles foron creados e escritos en ficheiro empregando a linguaxe de programación Matlab[4] e algunhas librarías incluídas neste programa de cálculo.

[4]MATLAB MathWorks - https://www.mathworks.com/products/matlab.html

O código está formado por tres programas, ou ficheiros de código fonte, diferentes. Todos eles implementan a operación de convolución 2D e foron desenvolvidos na seguinte orde:

- `conv.cpp` implementa o algoritmo en código C++ para a súa execución na CPU. Foi a primeira aproximación ao algoritmo da convolución e serviu como referencia da versión escrita en CUDA.
- `conv.cu` utiliza código CUDA para paralelizar o algoritmo e executalo na GPU. O resultado das execucións deste programa comparouse co resultado da execución na CPU coa finalidade de detectar e solucionar posibles erros na implementación.
- `conv_cudnn.cu` ademais de formar parte da toma de contacto con cuDNN, utilízase para comparar a interface de programación do noso programa coa da libraría.

Todos os programas teñen a mesma interface. Reciben dous parámetros de entrada, a ruta do ficheiro (*path*) da imaxe e do filtro, e escriben a imaxe resultado nun novo ficheiro.

```
./conv <image_file> <filter_file>
```

O funcionamento da versión secuencial e da versión CUDA é moi similar, mentres que a versión cuDNN presenta algunhas peculiaridades. Na estrutura das dúas primeiras podemos identificar os seguintes pasos:

1. Reserva de espazo na memoria para almacenar as entradas e a saída.
2. Lectura dos ficheiros da imaxe e do filtro.
3. Adición dun marco de ceros (*zero-pad*) ao redor da imaxe de entrada.
4. Execución do algoritmo da convolución 2D.
5. Escritura en ficheiro do resultado da operación.
6. Liberación da memoria.

Nos seguintes apartados coñeceremos as particularidades de cada implementación. Agora imos centrar a atención sobre unha función común a ambas, aquela que adiantábamos no terceiro punto: `zero_pad`. Como explicábamos no apartado 2.1.1 e visualizábamos na figura 2.1, a operación de convolución 2D reduce as dimensións das entradas, devolvendo unha imaxe máis pequena (en altura e largura) que a orixinal, mais co mesmo número de canles. Esta

```
   Data: A imaxe, as súas dimensións e as do padding
   Result: A nova imaxe co marco
 1 Calcular as dimensións da nova imaxe, newHeight e newWidth;
 2 for row ← 0 to newHeight do
 3 |   for col ← 0 to newWidth do
 4 |   |   for cha ← 0 to nChannels do
 5 |   |   |   if Existe image[row][col][cha] then
 6 |   |   |   |   *outPointer = image[row][col][cha];
 7 |   |   |   else
 8 |   |   |   |   *outPointer = 0;
 9 |   |   |   end
10 |   |   |   Incrementar outPointer;
11 |   |   end
12 |   end
13 end
```

Algoritmo 1: Función `zero_pad`

redución prodúcese porque durante a operación dos valores límite da imaxe o filtro sobresae da mesma, resultando imposible realizar o cálculo. En concreto, as dimensións da imaxe resultante son as seguintes:

$$P = (H - R)/Stride + 1$$
$$Q = (W - S)/Stride + 1$$

Onde P e Q son, respectivamente, a altura e a largura da imaxe resultante; H e W, teñen o mesmo significado para a imaxe de entrada e R e S para o filtro. En caso de engadir á operación un *stride* distinto de 1, este tamén afectaría á redución [3]. Como de momento non imos alterar o valor do *stride*, podemos simplificar o cálculo e afirmar que a redución, no noso caso, é de $R - 1$ filas e $S - 1$ columnas. Para evitar a perda de información que isto supón, no filtrado de imaxes adoita engadirse unha marxe á imaxe orixinal enmarcándoa, coma se dun cadro se tratase, con píxeles cuxo valor pode variar. En xeral, atopamos dous tipos de marcos: no primeiro todos os valores son cero (*zero-pad*), é o máis sinxelo pero nótanse bastante os seus efectos, escurecendo os bordes da imaxe resultante. No segundo, utilízanse os valores limítrofes da imaxe para dar valor aos do *padding*, polo que resulta máis discreto. A nosa función `zero_pad` emprega o primeiro destes marcos. A súa operación consiste en reservar memoria para aloxar a imaxe enmarcada e ir recorrendo ese novo espazo mentres copia nel, ou ben ceros no marco, ou ben os datos da imaxe no interior do mesmo. O Algoritmo 1 resume cun breve pseudocódigo a súa operativa.

O tamaño do *padding* varía en función do tamaño do filtro, como durante a operación colocamos o píxel de interese no centro do filtro, o tamaño do marco vai ser sempre a división

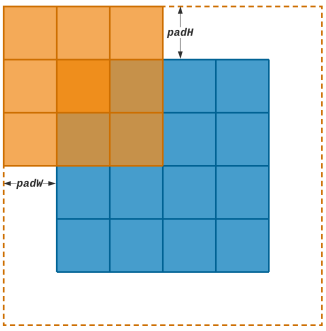


Figura 4.2: É necesario engadir un marco á imaxe orixinal.

enteira das súas dimensións entre 2. No esquema da figura 4.2, ademais de ilustrar a necesidade de engadir este marco, podemos comprobar que as dimensións do *padding* poden ser calculadas mediante as seguintes fórmulas:

$$padH = \left\lfloor \frac{R}{2} \right\rfloor \quad padW = \left\lfloor \frac{S}{2} \right\rfloor$$

Sendo R e S, como ata agora, a altura e a largura do filtro empregado. A imaxe enmarcada aumentará, polo tanto, $2 \cdot padH$ a súa dimensión vertical (altura, H) e $2 \cdot padW$ a súa dimensión horizontal (largura, W).

Unha vez realizadas as operacións comúns a ambos programas, é o momento de explicar as particularidades de cada un. Nos seguintes apartados veremos a implementación do algoritmo da convolución en C++ para a súa execución na CPU (4.1.1), a transformación deste programa en código CUDA para a súa execución na GPU (4.1.2) e, finalmente, unha breve demostración de como realizar a operación de convolución empregando cuDNN (4.1.3).

4.1.1 Execución na CPU - *conv.cpp*

O primeiro programa desenvolto nesta iteración, `conv.cpp`, implementa a convolución en C++ e se executa secuencialmente na CPU. O algoritmo é básico, no sentido de que non implementa ningunha optimización e simplemente codifica o procedemento que xa explicamos no apartado 2.1.1, figura 2.1. No pseudocódigo 2 mostramos unha versión simplificada da nosa implementación, preparada para procesar filtros de dimensións tanto pares como impares. Tal e como faríamos de forma gráfica, os límites dos bucles comezan colocando o filtro na posición $padH, padW$ da imaxe enmarcada, que se corresponde co primeiro valor da imaxe orixinal. A imaxe de saída calcúlase píxel a píxel, polo que só é necesario ir incrementando un punteiro que sinale á posición da imaxe de saída que estamos a calcular nese momento.

```
   Data: A imaxe, o filtro e as súas dimensións
   Result: A imaxe de saída
1  Calcular as dimensións do padding: padH e padW
2  for row ← padH to imgHeight − padH do
3      for col ← padW to imgWidth − padW do
4          for cha ← 0 to nChannels do
5              sum = 0;
6              for i ← −padH to padH do
7                  for j ← −padW to padW do
8                      sum += image[row+i][col+j][cha] * filter[padH+i][padW+j][cha];
9                  end
10             end
11             *outPointer = sum;
12             Incrementar outPointer;
13         end
14     end
15 end
```

Algoritmo 2: Convolución 2D. Código secuencial (CPU)

Por outro lado, se analizamos o patrón de acceso á memoria (figura 4.3), vemos que é secuencial para o filtro e a imaxe de saída, mais non o é no caso da imaxe de entrada. É necesario saber que en C as matrices gárdanse por filas e, como comentamos ao principio deste capítulo, a información da cor neste caso está intercalada. Non obstante, a pesar de que en xeral accedemos aos datos por filas, nos dous bucles máis internos percorremos os valores da imaxe aos que se superpón o filtro, sendo estes grupos de valores da largura do filtro e separados entre si. No caso da figura, procesamos tres píxeles (de tres valores cada un, un por canle) e realizamos un salto, outros tres píxeles e outro salto... así ata rematar o percorrido da imaxe.

Aínda que con esta primeira versión non estamos a procurar un rendemento excepcional, é interesante coñecer como afectan este tipo de detalles ao tempo de execución do programa. A localidade dos datos, por exemplo, é un parámetro moi importante que debemos coidar para minimizar a latencia dos accesos a memoria e, nalgúns casos, mellorala pode ser tan sinxelo como alterar a orde dalgún bucle.

4.1.2 Execución na GPU - *conv.cu*

O seguinte paso consiste en converter o código secuencial C++ en código CUDA, e executalo de forma paralela na GPU. A maiores do que xa fixemos no primeiro programa (reservar memoria, ler os ficheiros e engadir o *padding*), temos que reservar espazo na memoria do dispositivo e transferir os datos da imaxe e o filtro dende a memoria do *host*, tal e como vimos no capítulo 2, apartado 2.2.3. Para simplificar a xestión dos posibles erros que poidan xurdir durante a execución das funcións CUDA, definimos a seguinte *macro*:

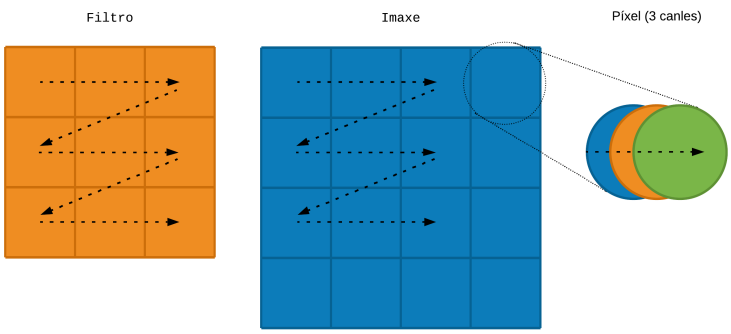


Figura 4.3: Orde dos accesos a memoria.

```
#define CUDA_SAFE_CALL(ans) {gpuAssert((ans), __FILE__, __LINE__);}
inline void gpuAssert(cudaError_t code, const char *file, int line)
{
   if (code != cudaSuccess) {
      fprintf(stderr, "GPUassert: %s (%s : %d)\n",
                           cudaGetErrorString(code), file, line);
      exit(code);
   }
}
```

Esta función comproba o valor de retorno das funcións CUDA, de tipo `cudaError_t`, e mostra por pantalla o tipo de erro detectado antes de deter a execución do programa. Envolver as funcións CUDA nesta *macro* permite detectar os erros que se poden producir durante a execución do programa. Unha vez transferidos os datos, podemos executar o *kernel*, neste caso, definimos mallas e bloques bidimensionais de tamaño configurable:

```
dim3 grid(16,16);
dim3 block(2,2);

convolution <<< grid, block >>>(
    output, input, filter, // entradas e saídas
    H, W, C, R, S          // dimensións
);
```

Tras a execución do *kernel* facemos unha nova transferencia, esta vez do resultado (`output`) á memoria do *host*, utilizando unha vez máis a función `cudaMemcpy` envolta na *macro* que comentamos antes. É importante destacar que o feito de ter que transferir os datos de entrada e os resultados da execución, antes e despois do *kernel*, produce un sobrecusto a maiores do tempo que lle leva á GPU realizar a operación. É por isto que só conseguiremos mellorar os

```
   Data: A imaxe, o filtro e as súas dimensións
   Result: A imaxe de saída
1  Calcular os identificadores de fío para o reparto e as dimensións da imaxe de saída;
2  for rowb ← idx * nrows to outHeight step nthreadsx*nrows do
3      for row ← rowb to rowb + nrows do
4          for colb ← idy * ncols to outWidth step nthreadsy*ncols do
5              for col ← colb to colb + ncols do
6                  for cha ← 0 to nChannels do
7                      sum = 0;
8                      for i ← 0 to filHeight do
9                          for j ← 0 to filWidth do
10                             sum += image[row+i][col+j][cha] * filter[i][j][cha];
11                         end
12                     end
13                     output[row][col][cha] = sum;
14                 end
15             end
16         end
17     end
18 end
```

Algoritmo 3: Convolución 2D. Código paralelizado (GPU)

resultados da CPU se dispoñemos do suficiente paralelismo e logramos explotalo. Finalmente, tras escribir a imaxe resultante en ficheiro e antes de liberar as memorias, comprobamos que o resultado desta execución coincide co do programa anterior. Para iso, lemos os datos dende o ficheiro que escribiu o anterior programa e percorremos cada posición das imaxes comprobando a súa igualdade.

A función da convolución tamén precisa dunha adaptación para ser executada na GPU. A primeira diferencia que se aprecia ao consultar o pseudocódigo 3 é o aumento no número de bucles, provocado pola estratexia de reparto do traballo empregada. Esta estratexia consiste en que cada fío calcula un número de filas (`nrows`) e de columnas (`ncols`) fixo, especificado no propio código da función, un traballo que se corresponde co segundo e o cuarto bucle do pseudocódigo, respectivamente. Cando un fío remata o seu bloque de datos, se aínda quedan píxeles por calcular, comeza a procesar outra rexión de datos distanciada da primeira, é dicir, inicia unha nova iteración do primeiro ou o terceiro bucle. Esta operación repítese de forma cíclica ata rematar con todos os datos da imaxe. O número de posicións que separan os bloques de datos e que teñen que avanzar os fíos en cada rolda de procesamento é de `nthreadsx*nrows` (ou `nthreadsy*ncols` no caso das columnas), é dicir, o número de filas ou columnas que calculou o fío polo número de fíos nesa dimensión do *grid*. No esquema da figura 4.4 representamos de forma máis clara este reparto de píxeles. Neste exemplo, hai un total de 16 fíos organizados nun *grid* de 2x2 bloques e 2x2 fíos por bloque. Identificámolos

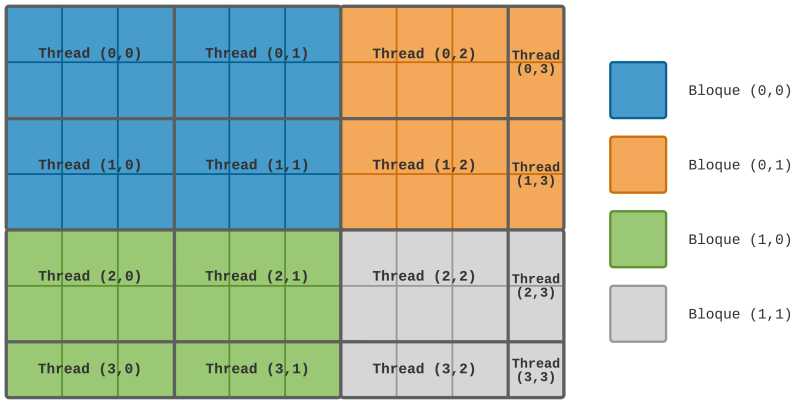


Figura 4.4: Superposición dos fíos do *grid* sobre os píxeles da imaxe.

polo seu ID global, é dicir, a nivel de *grid*. Este identificador ten dúas componentes que se calculan da seguinte forma:

```
1 int idx = blockDim.x * blockIdx.x + threadIdx.x;
2 int idy = blockDim.y * blockIdx.y + threadIdx.y;
```

Cada fío calcula 2 filas e 3 columnas, polo tanto, hai suficientes fíos para cubrir a imaxe sen necesidade de repetir, de feito, os últimos fíos teñen que calcular menos píxeles que o resto. Se unha vez repartidos os píxeles entre os fíos quedasen posicións por calcular, comezaría unha nova rolda de procesamento repetindo o mesmo patrón que na primeira.

Finalmente, é interesante ver como o tempo de execución do programa diminúe en gran medida grazas a esta paralelización. Agora somos capaces de executar convolucións con imaxes e filtros de maior tamaño e que leven incluso menos tempo.

4.1.3 Exemplo con cuDNN - *conv_cudnn.cu*

O último programa desenvolto para esta versión baséase no uso da API de cuDNN. Como comentamos no capítulo 2, trátase dunha libraría pensada para optimizar aquelas funcións que son máis empregadas en redes de aprendizaxe profunda. Por iso, o algoritmo da convolución que utilizamos neste programa non é o mesmo que implementamos nos dous anteriores, senón a versión para CNNs que víamos no apartado 2.1.2.

O primeiro que chama a atención ao usar cuDNN é que ofrece unha API baseada en contexto, o que, segundo o seu manual, permite unha fácil interoperabilidade entre múltiples fíos e tamén cos *Streams* de CUDA. Como ilustra a figura 4.5 todos os programas teñen que definir

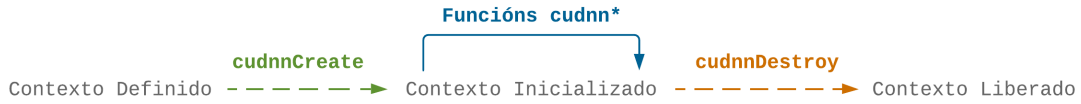


Figura 4.5: Ciclo de vida do contexto cuDNN

un contexto de tipo `cudnnHandle_t`, inicializalo, pasarllo as funcións que necesiten executar e, finalmente, liberalo ao finalizar o programa. O funcionamento deste programa consta dos seguintes pasos:

1. Declaración e inicialización dos descritores da imaxe de entrada, do filtro e da operación de convolución.

   Os descritores son variables de tipo `cudnn*Descriptor_t`. Dependendo do obxectivo, substituiremos `*` por `Tensor`, `Filter` ou `Convolution`. A inicialización lévase a cabo a través de funcións co seguinte patrón de nomeado: `cudnnCreate*Descriptor` e `cudnnSet*Descriptor`. A primeira inicializa o descritor e a segunda permite establecer as súas propiedades: dimensións, tipo de dato e disposición dos datos no caso das imaxes (tensores) e, no caso da convolución, os parámetros *padding*, *stride* e a dilatación.

2. Cálculo das dimensións da imaxe de saída, declaración e inicialización da mesma.

   Da mesma forma que nós calculábamos as dimensións da imaxe de saída para o noso algoritmo, a función `cudnnGetConvolution2dForwardOutputDim` obtén estes datos a partir dos descritores da imaxe, o filtro e a convolución. Utilizamos estas dimensións para inicializar o descritor do tensor de saída.

3. Cálculo do mellor algoritmo para os datos descritos e das dimensións do espazo de traballo necesario para dito algoritmo.

   A función `cudnnGetConvolutionForwardAlgorithm` elixe o algoritmo máis óptimo a partir dos descritores que definimos anteriormente. Por outro lado, coa función `cudnnGetConvolutionForwardWorkspaceSize` obtemos o número de bytes que ocupa o espazo de traballo que necesita dito algoritmo para funcionar.

4. Reserva de espazo na memoria do dispositivo, inicialización da imaxe e o filtro e copia dos seus valores dende a memoria do *host*. Tal e como víamos nos programas anteriores.

5. Execución da operación de convolución, `cudnnConvolutionForward`. Na documentación de cuDNN[5] noméanse tamén os algoritmos `cudnnConvolutionBackwardData` e `cudnnConvolutionBackwardFilter`, mais estas funcións non realizan a convolución,

[5] cuDNN Developer Guide: Convolution Functions - https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html#tensor-ops-conv-functions

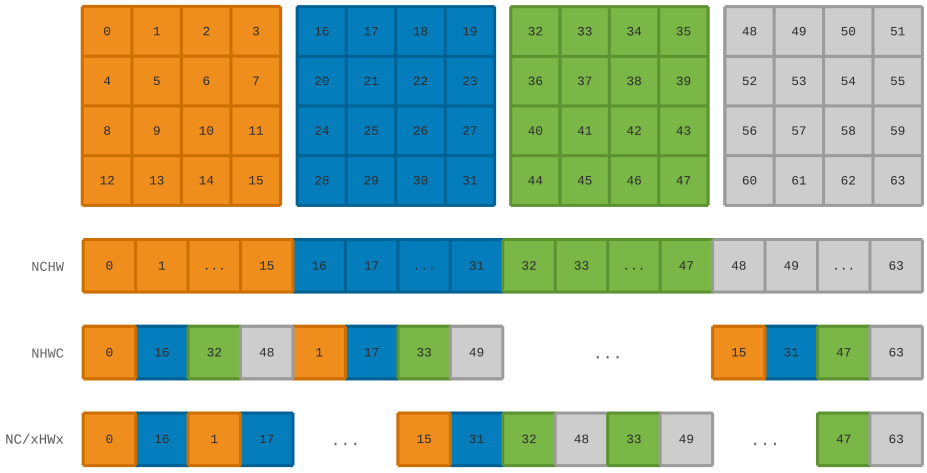


Figura 4.6: Distintos formatos de almacenamento dos datos dunha imaxe en memoria.

senón que calculan os gradientes utilizados na propagación cara atrás (*backpropagation*) [22] das redes neuronais.

6. Copia do resultado á memoria do *host*, escritura do mesmo en ficheiro e liberación de recursos, incluíndo a memoria, os descritores e o propio contexto cuDNN.

Como comentábamos ao comezo da sección 4.1, o noso programa só admite un tipo de disposición dos datos en memoria, *NHWC* (ver táboa 2.1), en cambio as funcións de cuDNN permiten ata 3 posibles formas de organizar as dimensións dos datos: *NCHW*, *NHWC* e *NC/xHWx*. Podemos velas representadas na figura 4.6. Á hora de inicializar os descritores temos que indicar o formato no que están gardados en memoria través do enumerado `cudnnTensorFormat_t`, que pode tomar os seguintes valores:

- `CUDNN_TENSOR_NCHW`: Indica que os datos da imaxe ou do filtro están gardados en memoria na seguinte orde: tamaño de *batch*, número de canles, altura e largura.
- `CUDNN_TENSOR_NHWC`: Formato usado nos anteriores programas. Indica que os datos están gardados na seguinte orde: tamaño de *batch*, altura, largura e, por último, número de canles ou mapas de características.
- `CUDNN_TENSOR_NCHW_VECT_C`: Este formato, tamén chamado *NC/xHWx*, agrupa as canles en grupos de x e cada grupo almacénase en formato *NHWC*. Só pode usarse cando o tipo de dato indicado é un vector.

Unha vez realizado este traballo de inmersión na API de cuDNN, semellaba necesario redeseñar os nosos programas para achegarnos á interface ofrecida por esta libraría. Na seguinte

iteración, realizaremos unha aproximación a esta interface definindo estruturas de datos que conteñan as dimensións das matrices e funcións que simplifiquen os procedementos máis habituais, mais sen diferenciar como fai cuDNN entre os descritores das imaxes e dos filtros, xa que ambos poden ser descritos como tensores. Ademais, os programas ata o momento non implementan o algoritmo que utilizan as redes neuronais convolucionais, senón unha convolución 2D sinxela, empregada en procesamento de imaxes para realizar filtrados espaciais. O seguinte paso é, polo tanto, implementar o algoritmo empregado polas capas convolucionais para realizar a convolución en lotes.

4.2 Convolución directa en lotes (v2.0)

Nesta versión realizamos por primeira vez a operación de convolución en lotes. Esta operación implementa nunha mesma función a convolución de varias imaxes cun mesmo conxunto de filtros. O noso programa realiza dúas implementacións do algoritmo, unha función en C++ que se executa de forma secuencial na CPU, e un *kernel* CUDA que se paraleliza na GPU. Tamén engade a medición dos tempos de execución de ambas funcións, co obxectivo de comparar a diferenza no rendemento. O código desta versión está sinalado mediante a etiqueta `v2.0` no repositorio[6].

Na sección explicaranse as funcionalidades engadidas nesta segunda versión do código, comezando pola nova interface baseada en estruturas de datos (apartado 4.2.1), e comentaranse os cambios no algoritmo (apartado 4.2.2) que pasa de ser un filtrado de imaxes a realizar a convolución tal e como é comprendida no ámbito das redes neuronais convolucionais.

4.2.1 Estruturas de datos

Algo que destaca nesta versión é que intenta seguir o exemplo da API de cuDNN no referido á interface de programación. Con este obxectivo foron definidos varios *structs* que almacenan a información das matrices e funcións para inicializar os seus valores. Sendo coherentes coa terminoloxía matemática empregada nas aplicacións de aprendizaxe profunda, en adiante falaremos de tensores para referirnos as matrices multidimensionais involucradas nas operacións. No eido das matemáticas, un tensor é unha entidade alxébrica de varias compoñentes (dimensións) que pode representar tanto escalares como vectores ou matrices, sendo independente de calquera sistema de coordenadas. As estruturas implementadas e as funcións vinculadas a estas son as seguintes:

- Por un lado, a información das imaxes de entrada e saída e dos filtros gárdase en estruturas de tipo `tensorDescriptor_t`, nas que indicamos a disposición en memoria dos

[6] Etiqueta *v2.0* - https://git.fic.udc.es/s.aguado/tfg/tree/v2.0

datos (un valor do enumerado `tensorFormat_t` que pode ser ou ben *NCHW*, ou ben *NHWC*) e as catro dimensións da matriz: alto, largo, profundidade (número de canles ou mapas de características) e tamaño do lote ou *batch*. Para cubrir ditos datos utilizamos a función `setTensorDescriptor`.

- A convolución tamén dispón da súa propia estrutura, `convolutionDescriptor_t`, que indica os típicos parámetros: *padding*, *stride* e dilatación. A maiores inclúe o modo de operación (`convolutionMode_t`) que normalmente é *CONVOLUTION* pero, se quixéramos invertir o filtro, teríamos que establecelo a *CROSS_CORRELATION*. Non obstante, aínda non é posible cambiar o modo de operación nas funcións desenvolvidas ata o momento. Ademais, dispoñemos doutra función `setConvolutionDescriptor` para encher os campos desta estrutura, similar á implementada para os tensores.
- Por outro lado, a función `setColvolutionOutputDescriptor` completa os campos do descritor do tensor de saída a partir dos descritores dos tensores de entrada (as imaxes e o filtro) e dos da convolución. Desta forma, o programa principal non ten que responsabilizarse de ningún cálculo a maiores.
- Finalmente, o enumerado `convolutionAlgorithm_t` permitirá, cando dispoñamos de varios, seleccionar o algoritmo da convolución desexado. O obxectivo é que a función `getConvolutionAlgorithm` obteña, en base aos descritores definidos previamente, o algoritmo máis axeitado. Este valor é necesario para calcular, con axuda da función `getConvolutionWorkspaceSize`, o tamaño do espazo de traballo que necesita dito algoritmo para executarse.

4.2.2 Algoritmos

O obxectivo deste apartado é subliñar os aspectos relevantes das principais funcións implementadas ata o momento, e observar tamén as diferenzas destas respecto as desenvolvidas previamente. Aínda que foron elaborados varios algoritmos en C++, en función de se os datos estaban en formato *NHWC*, como na versión anterior, ou en formato *NCHW*; finalmente seleccionamos o que semellaba máis axeitado para utilizar como referencia. A partir desta función secuencial implementamos unha versión paralelizada en linguaxe CUDA, seguindo o mesmo método que xa empregamos anteriormente.

A diferenza da versión 1.0, na que foron implementados dous programas por separado (tres, tendo en conta a versión cuDNN), desta vez as funcións son executadas no mesmo programa e utilizando os mesmos datos. Isto é necesario para asegurar unha correcta medición dos tempos de execución e poder facer unha comparación xusta dos resultados. Ademais, nesta versión cambia o propio algoritmo da convolución, que anteriormente consistía no filtrado

```
Data: A imaxe, o filtro e todos os descritores
Result: A imaxe de saída
for n ← 0 to N do
    for k ← 0 to K do
        for c ← 0 to C do
            for p ← 0 to P do
                for q ← 0 to Q do
                    if c == 0 then output[n][k][p][q] = 0 ;
                    for r ← 0 to R do
                        for s ← 0 to S do
                            output[n][k][p][q] += input[n][c][p+r][q+s] * filter[k][c][r][s];
                        end
                    end
                end
            end
        end
    end
end
```

Algoritmo 4: Convolución por lotes. Código secuencial (CPU)

```
Data: A imaxe, o filtro e todos os descritores
Result: A imaxe de saída
Cálculo dos límites dos bucles de cada fío, n_ini, k_ini e n_fin, k_fin;
for n ← n_ini to min(N, n_fin) do
    for k ← k_ini to min(K, k_fin) do
        for c ← 0 to C do
            for p ← 0 to P do
                for q ← 0 to Q do
                    if c == 0 then output[n][k][p][q] = 0 ;
                    for r ← 0 to R do
                        for s ← 0 to S do
                            output[n][k][p][q] += input[n][c][p+r][q+s] * filter[k][c][r][s];
                        end
                    end
                end
            end
        end
    end
end
```

Algoritmo 5: Convolución por lotes. Código paralelizado (GPU)

dunha imaxe en tres dimensións empregando un único filtro, e agora realiza a operación da convolución que xa explicamos no apartado 2.1.2. Esta é a operación que realizan repetidamente as capas convolucionais das redes de aprendizaxe profunda e nela interveñen matrices multidimensionais chamadas tensores, habitualmente de 4 dimensións: alto, largo, profundidade e tamaño de lote. Estas matrices almacenan as características extraídas das imaxes durante as sucesivas capas convolucionais. Cada nivel de profundidade (correspondente coa terceira dimensión da matriz, a que nunha imaxe almacenaría as canles coa información da cor) constitúe un mapa de características. A característica diferenciadora deste algoritmo, ademais da adición dunha cuarta dimensión ás imaxes, é que se suman os resultados parciais da convolución de cada mapa de característicasw co seu respectivo filtro. Para estes efectos, o corpo dos bucles inclúe unha condición que establece a cero o valor do píxel de saída cando se está a computar a primeira canle. Sobre este píxel, sen necesidade de apoiarse en ningunha variable auxiliar, calcúlase a suma acumulativa do produto dos valores da xanela que está a cubrir o filtro, para esa primeira canle e para todas as restantes.

Nos pseudocódigos 4 e 5 podemos apreciar que a única diferenza entre o algoritmo secuencial e o paralelizado son os límites dos dous bucles máis externos, ademais do feito de ter que calcular ditos límites a partir dos identificadores de fío. As entradas a ambos algoritmos son os datos da imaxe e do filtro, en formato NCHW, e as súas dimensións. Sobre eles non se realiza ningunha transformación previa. Respecto ao reparto do traballo entre os distintos fíos de execución, nesta ocasión decidimos dividir o procesamento das imaxes entre os bloques e, dentro de cada bloque, asignarlle un filtro a cada *thread*; a diferenza da versión anterior na que repartíamos o conxunto de datos en bloques de menor tamaño e de forma cíclica. Así mesmo, empregamos un *grid* unidimensional, indicando só a compoñente x na chamada ao *kernel*. Por tanto, podemos deducir que para aproveitar ao máximo o paralelismo cómpre lanzar tantos bloques como imaxes hai no lote, e tantos fíos por bloque como filtros. Isto vai supoñer varias limitacións, comezando polo nivel de paralelismo que podemos explotar, pero tamén á hora de utilizar a memoria local. En versións posteriores foi mellorado este reparto, nas seguintes ecuacións podemos ver como se realiza na iteración actual:

$$n\_lim = \left\lceil \frac{N}{\texttt{gridDim.x}} \right\rceil \qquad n\_ini = \texttt{blockIdx.x} \cdot n\_lim \qquad n\_fin = n\_ini + n\_lim$$

$$k\_lim = \left\lceil \frac{K}{\texttt{blockDim.x}} \right\rceil \qquad k\_ini = \texttt{threadIdx.x} \cdot k\_lim \qquad k\_fin = k\_ini + k\_lim$$

Onde *n_lim* e *k_lim* son o número de imaxes ou filtros que debe calcular cada bloque ou fío, respectivamente; *n_ini* e *k_ini* o índice do primeiro elemento dese bloque ou fío; e final-

mente, *n_fin* e *k_fin* representan o límite superior de cada un dos bucles paralelizados.

No capítulo 5, comentaremos o resultado da execución deste algoritmo confrontando a versión secuencial e a paralelizada. Unha vez máis, o rendemento da implementación CUDA verase afectado pola latencia das transferencias de datos dende a memoria do *host* á memoria do dispositivo. Así e todo, se dispoñemos do suficiente paralelismo de datos, obteremos mellores resultados coa versión concorrente.

4.3 Algoritmo GEMM (v3.0)

Con esta versión buscamos introducir unha mellora no rendemento do programa que non dependa só da paralelización do algoritmo. Con ese obxectivo, substituímos o algoritmo orixinal por unha versión mellorada chamada GEMM (do inglés, *General Matrix Multiply*) que xa explicamos no capítulo 2, apartado 2.1.3. Como comentamos en dito apartado, o algoritmo GEMM divide a operación de convolución en dúas: unha reordenación e replicado dos datos de entrada e unha multiplicación de matrices. No afán de atopar a implementación máis óptima, foron realizadas dúas aproximacións distintas: unha na que as dúas operacións se realizan nun único *kernel* e outra que divide o algoritmo en dous *kernels* distintos, un para a reordenación e outro para a multiplicación de matrices. Todas as funcionalidades comentadas nesta sección poden ser consultadas na rama `feature/gemm`[7] do repositorio.

Antes de nada, é necesario explicar que para universalizar o reparto do traballo entre os fíos e calcular os límites dos bucles utilizados nos algoritmos, extraeuse dito cálculo nunha función do dispositivo (marcada coa directiva `__device__`) que reutilizan varios dos *kernels*. Esta función contempla dous escenarios:

- Que haxa máis elementos que fíos, polo que un fío terá que calcular varios elementos.
- Que haxa máis fíos que elementos, neste caso asignaremos varios fíos a un único elemento. Se podemos, seguiremos dividindo o procesamento de dito elemento (unha imaxe, fila, columna...) entre os fíos asignados. Se non é posible realizar máis divisións (un píxel, un elemento indivisible) quedarán fíos sen ocupar.

A función, chamada `getLimits`, recibe como parámetros: o número de fíos (ou unidades de procesamento), o número de elementos a dividir e o índice absoluto do fío que está a executar a función. As saídas desta función son: o número de elementos que ten que calcular cada fío, o índice relativo do fío dentro do bloque de procesamento e os límites inferior e superior

[7] Rama *feature/gemm* - https://git.fic.udc.es/s.aguado/tfg/tree/v3.0

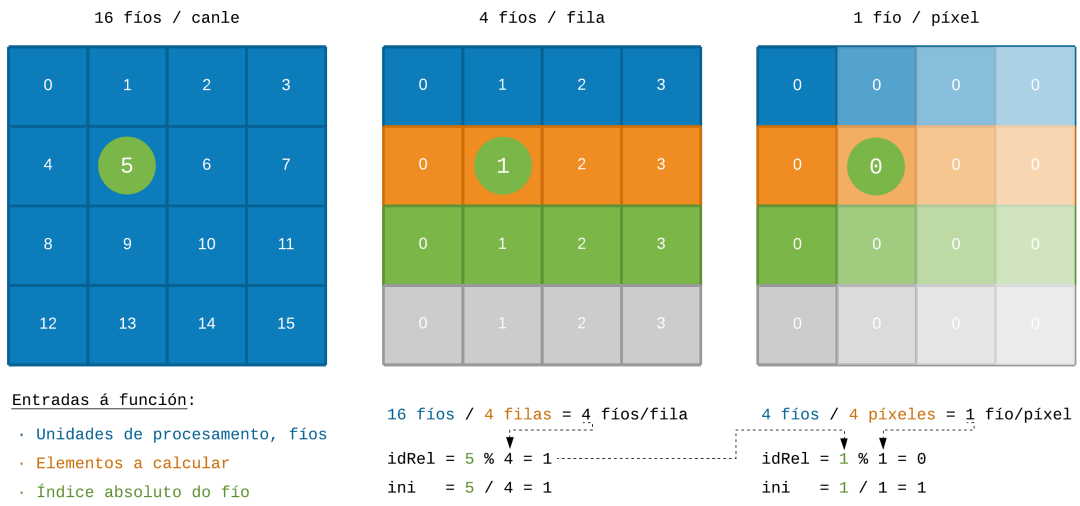


Figura 4.7: Reparto de elementos realizado pola función `getLimits`.

dos elementos que debe calcular dito fío. O esquema da figura 4.7, pretende exemplificar este reparto. No exemplo dividimos unha matriz de 4x4 elementos entre 16 fíos, para iso, chamamos á función `getLimits` dúas veces: unha para dividir as filas entre o número total de fíos e outra para dividir as columnas entre os fíos que sobraron na anterior división. Trala primeira chamada obtemos unha división lóxica en 4 bloques de 4 *threads*, onde cada bloque procesa unha única fila. O índice relativo (`idRel`) de cada fío dentro do seu respectivo bloque lóxico utilizámolo na segunda chamada como índice absoluto, este índice serve para identificar a primeira posición (`ini`) que lle toca calcular ao fío. Nesta segunda chamada obtemos 16 bloques lóxicos formados por un único fío.

En definitiva, coa función que acabamos de explicar podemos obter os límites dos bucles de forma individualizada para cada un dos fíos que executan o *kernel*. Por tanto, agora que sabemos paralelizar os bucles, imos explicar as funcións que os conteñen.

En primeiro lugar, como dicíamos ao comezo da sección, para converter a convolución nunha multiplicación de matrices antes precisamos reordenar, e replicar ao mesmo tempo, os datos da imaxe de entrada. Esta reordenación, da que xa falamos no capítulo 2 e exemplificamos agora co esquema da figura 4.8, consiste en ir recorrendo a imaxe orixinal seguindo o mesmo camiño que cando desprazamos o filtro sobre ela para realizar a convolución directa, pero en lugar de multiplicar os seus valores polos do filtro, copiámolos a outra rexión da memoria colocándoos da forma que podemos ver na figura. O resultado desta operación é unha matriz bidimensional que supera de forma considerable o tamaño da imaxe orixinal, así que para aloxar esta matriz necesitaremos reservar a maiores un espazo de traballo (*workspace*) na

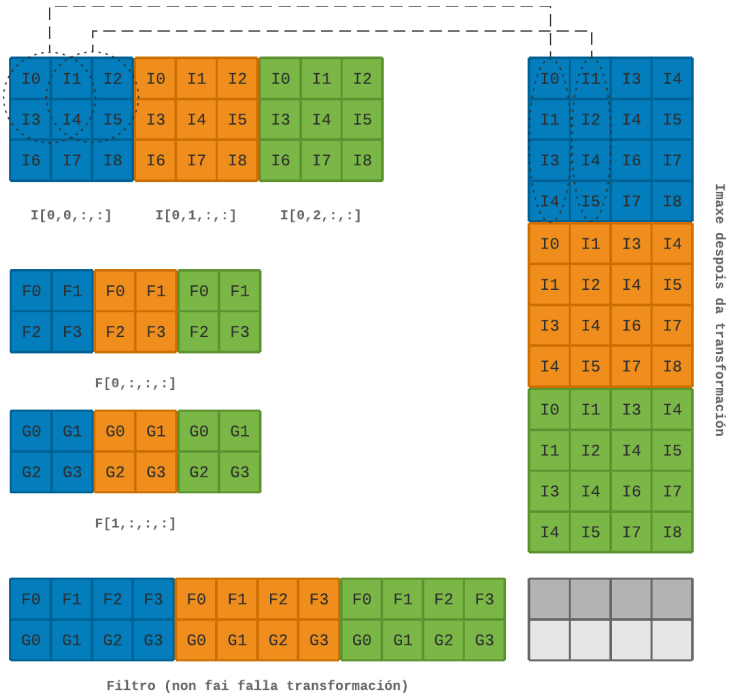


Figura 4.8: Transformación dunha imaxe de dimensións *1x3x3x3* (*NxCxHxW*) e dous filtros de tamaño *2x3x2x2* (*KxCxRxS*) para o algoritmo GEMM.

memoria. Podemos supoñer, polo tanto, que o nivel de ocupación da memoria vai ser un dos puntos críticos deste algoritmo. Para levar a cabo esta transformación definimos, nun principio, unha función do dispositivo chamada `im2col` (pseudocódigo 6), que posteriormente convertemos nun *kernel*. A orde dos bucles no pseudocódigo 6 non é casual, foi especificamente alterada para favorecer a localidade dos datos na medida do posible, e acceder de forma secuencial á memoria do espazo de traballo.

En segundo lugar, a outra función que necesitamos é unha multiplicación de matrices. Na función `matrixProduct` do pseudocódigo 7 podemos ver que as matrices implicadas son o filtro e o espazo de traballo (*workspace*) que contén a imaxe transformada. As dimensións da imaxe transformada, recordamos, son $R \cdot S \cdot C$ filas e $P \cdot Q$ columnas, sendo R e S as dimensións do filtro, C o número de mapas de características (canles) e P e Q as dimensións da imaxe de saída. A orde dos bucles, unha vez máis, está pensada para realizar as lecturas e escrituras por filas, de forma que se aproveite a localidade dos datos. Ambas funcións procesan unha única imaxe tridimensional, son despois os *kernels* os que xestionan o lote de imaxes. Nesta ocasión, foron realizadas dúas aproximacións do algoritmo.

```
Data: A imaxe, as súas dimensións e as dimensións do filtro
Result: A imaxe transformada contida no espazo de traballo
for c ← 0 to C do
    for r ← 0 to R do
        for p ← 0 to P do
            for q ← 0 to Q do
                for s ← 0 to S do
                    workspace[c][r*S + s][p*Q + q] = input[c][p+r][q+s];
                end
            end
        end
    end
end
```

Algoritmo 6: Función `im2col`. Transformación da imaxe de entrada.

```
Data: A imaxe transformada, o filtro, e as dimensións de ambos
Result: A imaxe de saída
for k ← 0 to K do
    for l ← 0 to R * S * C do
        for m ← 0 to P * Q do
            output[k][m] += filter[k][l] * workspace[l][m];
        end
    end
end
```

Algoritmo 7: Función `matrixProduct`. Multiplicación de matrices.

A primeira está formada por un único *kernel* que realiza ambas operacións. O problema desta implementación é que está moi condicionada pola configuración do *grid*, é dicir, o número de bloques e fíos indicados no momento do lanzamento do *kernel*. A razón é que, antes de comezar a multiplicación do *workspace* e do filtro, os fíos implicados deben agardar a que os demais rematen a reordenación dos datos da imaxe, como é lóxico, e a consecuente escritura do espazo de traballo. Isto non suporía ningún problema se dispuxésemos dunha ferramenta para facelo dende o interior do *kernel*, a cuestión é que a única función de sincronización accesible dende o código do dispositivo é `__syncthreads()`, que sincroniza os fíos só a nivel de bloque. En consecuencia acabaremos asignando unha única imaxe a cada bloque, infrautilizando nalgúns casos as capacidades da GPU. Podemos resumir o funcionamento desta función nas seguintes instrucións:

```
Para cada imaxe "n" no lote, un bloque executa:
    im2col(imaxe[n], filtro, workspace[n])
    __syncthreads() // espera polos fíos deste bloque
    matrixProduct(filtro, workspace[n], output[n])
```

A segunda aproximación en cambio, lanza dous *kernels*, un para cada función, que apro-

veitan todos os fíos que foron lanzados. Ademais, é posible optimizar a configuración de lanzamento segundo as necesidades de cada función. Entre ambos *kernels* tamén é necesario sincronizar os fíos, pola mesma razón, mais esta vez realízase dende o código do *host* coa función `cudaDeviceSynchronize()`. Desta forma faríamos:

```
Dende o código da CPU lanzamos dous kernel:
    im2col <<< grid,block >>>(input, filter, workspace);
    cudaDeviceSynchronize();
    matrixProduct <<< grid,block >>>(output, filter, workspace);
```

Esta é a versión que deu mellores resultados na maioría dos casos, ademais, tamén é a opción escollida pola libraría cuDNN. Podemos comprobalo observando os resultados do perfilado dun programa cuDNN que execute o algoritmo GEMM.

Finalmente, tras analizar o rendemento do algoritmo e realizar un perfilado da súa execución, consideramos que era posible introducir unha mellora no rendemento acelerando os accesos á memoria dos elementos que máis se reutilizan, neste caso, os filtros. Se centramos a atención no funcionamento do algoritmo, é fácil decatarse de que todos os filtros son usados con todas as imaxes, polo que os seus elementos son lidos en maior número que calquera outro. Reducindo un pouco a latencia de cada lectura, como esta vai ser repetida en múltiples ocasións, a mellora global acabará sendo notable. Trataremos con máis calma esta optimización no apartado 4.3.2. Non obstante, primeiro introduciremos unha mellora máis sinxela, pero non menos importante, nas transferencias entre o *host* e o dispositivo (apartado 4.3.1).

4.3.1 *Pinned Memory* (v3.1)

Como adiantábamos no capítulo 2 e comprobamos na primeira versión CUDA, o prezo a pagar por realizar o cómputo na GPU son as transferencias de datos do *host* ao dispositivo. Comentábamos anteriormente que a memoria reservada dende a CPU é por defecto paxinable e, polo tanto, inaccesible para a GPU. Cando chamamos á función `cudaMemcpy` sobre datos aloxados en páxinas de memoria, CUDA reserva un espazo de intercambio na *Pinned Memory*, onde a memoria non se pode paxinar, e dende aí copia os datos á GPU.

Podemos aforrar a primeira destas copias se dende o principio aloxamos os datos na *Pinned Memory*. Para xestionar a reserva deste tipo de memoria, CUDA expón as funcións `cudaMallocHost` e `cudaFreeHost`. A mellora introducida nesta versión non require de máis acción que substituír as chamadas ás funcións de C estándar `malloc` e `free`, que se usaban nun principio, polas outras dúas. Todos os cambios poden ser consultados no ficheiro

main.cu[8] da rama feature/pinned-memory[9] do repositorio.

4.3.2 *Shared Memory* (v3.2)

A última versión desenvolta neste traballo tiña como obxectivo reducir ao mínimo a latencia dos accesos á memoria onde se aloxan os filtros involucrados na convolución. Este algoritmo ten a característica de reutilizar en múltiples ocasións cada elemento do filtro, por iso, se fose posible recortar unha pequena parte do tempo que lle leva á GPU ler cada elemento do filtro, no conxunto da operación lograríamos percibir certa mellora.

Con este obxectivo presente, dispoñémonos a realizar unha versión alternativa dos algoritmos comentados na sección 4.3 onde, antes de realizar a multiplicación de matrices, copiamos os datos do filtro na *Shared Memory* [23]. Como o tamaño do filtro é un valor coñecido en tempo de execución, debemos reservar a memoria de forma dinámica e, como comentamos no capítulo 2 apartado 2.2.3, indicar nun terceiro parámetro da chamada ao *kernel* o número de bytes que ocupan os datos do filtro. É importante aproveitar que esta memoria é compartida a nivel de bloque, e que todos os fíos de dito bloque colaboren no proceso de copia, desta forma minimizaremos o impacto dunha acción que non deixa de ser engadida. O procedemento é o mesmo para as dúas aproximacións que comentamos anteriormente:

```
extern __shared__ int shared_filter[];
for (int i = ini; i < fin; i++)
  shared_filter[i] = filter[i];

__syncthreads(); // espera polos fíos deste bloque
```

Onde `ini` e `fin` son dúas variables que delimitan o número de elementos a copiar por cada fío do bloque. A única diferenza entre ambas implementacións explicarémola a continuación.

Ao empregar a *Shared Memory* temos que ter coidado co tamaño dos datos, o motivo é que esta memoria é moito máis escasa que a memoria global do dispositivo. Por exemplo, o computador do laboratorio (do que falaremos no capítulo 5) cunha tarxeta NVIDIA GeForce RTX 2080 Ti, dispón de 49152 bytes (48 KB) de memoria compartida. Este é xusto o tamaño que ocupa un único filtro de 64x64 elementos e 3 mapas de características; ou o que ocupan catro filtros de 32x32 e tamén 3 canles. Asumindo que cada elemento do filtro require 4 bytes

[8] Ficheiro *main.cu* - https://git.fic.udc.es/s.aguado/tfg/tree/v3.1/src/main.cu
[9] Rama *feature/pinned-memory* - https://git.fic.udc.es/s.aguado/tfg/tree/v3.1

de memoria:

$$1 \cdot 3 \cdot 64 \cdot 64 \ \ elementos \cdot 4 \, bytes = 49152 \, bytes$$
$$4 \cdot 3 \cdot 32 \cdot 32 \ \ elementos \cdot 4 \, bytes = 49152 \, bytes$$

Tendo en conta que non é para nada complicado atopar filtros deste tamaño en aplicacións de *Deep Learning* e de feito se utilizan bastante, esta limitación é bastante problemática. Para intentar afrontala, no *kernel* da segunda aproximación que executa a función *matrixProduct*, en lugar de copiar todos os filtros na *Shared Memory*, cada bloque copia un único filtro e realiza todas as operacións dese filtro co conxunto de imaxes. Desta forma podemos empregar filtros de maior tamaño sen esgotar a memoria compartida. Por desgraza, máis tarde descubriremos que as modificacións necesarias para levar isto a cabo terán efectos adversos no rendemento xeral do programa.

Comentaremos as probas realizadas sobre as distintas versións vistas ao longo deste capítulo, e as conclusións acadadas tras realizar a análise dos resultados, no capítulo 5: Probas e resultados.

4.3.3 cuBLAS (v3.2)

Unha vez feitas as probas das distintas versións GEMM ata o momento, decatámonos de que o rendemento acadado non é o que esperábamos. Co obxectivo de comprobar que o defecto deriva dunha implementación excesivamente simple (no sentido de pouco optimizada) da multiplicación de matrices, substituímos o *kernel* `matrixProduct_kernel` da segunda aproximación (chamada *Split* nas probas) que realiza dita operación por unha función da libraría cuBLAS. Concretamente, a función utilizada foi `cublasSgemm`, que executa a mesma operación incluíndo certas optimizacións. Esta función procesa unha única imaxe de cada vez, polo que, para que esta versión sexa equiparable as anteriores, facemos unha chamada á función `cublasSgemm` con cada imaxe do *batch*, e desta forma procesamos todas as imaxes do lote.

Os resultados desta operación serán analizados unha vez máis no capítulo 5, onde os compararemos cos da segunda aproximación do algoritmo (*Split*) que emprega a nosa multiplicación de matrices. No citado capítulo, poderemos ver representada graficamente a mellora que supoñen as optimizacións da libraría cuBLAS no algoritmo da convolución GEMM.

Capítulo 5

Probas e resultados

Ao remate de cada versión citada no capítulo 4, realizouse unha tarefa de análise dos resultados a nivel, tanto de corrección das operacións implementadas, como de rendemento dos propios algoritmos. Neste último capítulo, polo tanto, expoñeranse as probas realizadas e os resultados obtidos en ditas probas. Antes de proceder coa devandita análise, na sección 5.1 comentaremos as características do entorno de execución, incluíndo as especificacións dos equipos empregados neste traballo. Deseguido, na sección 5.2, comentaremos versión a versión os resultados da execución en cada plataforma.

5.1 Contorno de probas

Como mencionamos recentemente, resulta de gran importancia coñecer as características físicas (*hardware*) dos dispositivos, neste caso das GPUs, que executan os programas para poder analizar de forma obxectiva o seu funcionamento. Especificacións como o tamaño da memoria, das cachés, a *CUDA Compute Capability* da GPU ou as dimensións máximas que pode manexar do *grid*, son valores a ter en conta á hora de establecer a configuración de lanzamento do *kernel* ou incluso, dependendo das aplicacións, á hora de deseñar a propia implementación do mesmo. Dedicaremos o apartado 5.1.1 a comentar as dúas plataformas sobre as que foron executadas as probas, e o apartado 5.1.2 a mencionar as ferramentas utilizadas na recollida e análise dos datos.

5.1.1 Características dos equipos

Neste proxecto foron empregadas dúas plataformas distintas: un computador persoal portátil cunha tarxeta gráfica NVIDIA GeForce 840M de gama media-baixa, e un equipo de sobremesa cunha NVIDIA GeForce RTX 2080 Ti[1] de gama media-alta situado no laboratorio de investigación e accesible dende Internet mediante SSH. As diferencias entre ambas tarxetas

[1] GeForce RTX 2080 Ti - https://www.nvidia.com/es-es/geforce/graphics-cards/rtx-2080-ti/

Variable	GeForce 840M	GeForce RTX 2080 Ti
Arquitectura	Maxwell	Turing
CUDA Capability	5.0	7.5
Frecuencia da GPU	1124 MHz (1.12 GHz)	1620 MHz (1.62 GHz)
Frecuencia da memoria	1001 MHz	7000 MHz
Global Memory	2004 MB (2 GB)	10984 MB (11 GB)
Constant Memory	65536 Bytes	65536 Bytes
Shared Memory	49152 Bytes	49152 Bytes
Rexistros por bloque	65536	65536
Caché de nivel 2 (L2)	1048576 Bytes	5767168 Bytes
Multiprocesadores · Cores/MP	3·128 = 384 cores	68·64 = 4352 cores
Threads por MP	2048	1024
Threads por bloque	1024	1024

Táboa 5.1: Diferencias arquitectónicas entre as dúas GPUs empregadas no traballo.

gráficas poden ser consultadas na táboa 5.1, onde se recollen aquelas variables que tiveron maior relevancia para o proxecto. A nivel xeral, podemos dicir que estamos ante unha arquitectura de gama baixa (GeForce 840M), orientada a dispositivos portátiles, e outra de gama media-alta (GeForce RTX 2080 Ti) de escritorio. A diferencia entre ambas podemos apreciala no número de cores e na frecuencia dos procesadores e da memoria. Finalmente, é interesante coñecer que o equipo do laboratorio ten un procesador *AMD Ryzen Threadripper 1950X* de 16 núcleos e unha memoria RAM de 65GB, mentres que o do equipo portátil é un *Intel Core i7-4710MQ* de 4 núcleos e conta con 12GB de memoria.

5.1.2 Ferramentas de análise

O *CUDA Toolkit* dispón de varias ferramentas para recoller e visualizar os resultados da execución dos programas CUDA. Neste proxecto foron utilizadas dúas: o comando `nvprof`[2] e a aplicación de escritorio NVIDIA Nsight Compute[3]. Ambos son *profilers*, é dicir, ferramentas para caracterizar o rendemento das diferentes partes do código e desta forma detectar as áreas onde é posible levar a cabo una optimización do rendemento. A diferencia entre ambos, é que Nsight Compute é unha ferramenta con interface gráfica, coa que podemos visualizar o grao de ocupación da GPU ou os accesos ás distintas memorias, entre outras cousas. Este programa pretende substituír á versión gráfica de `nvprof`, *Visual Profiler*, nas próximas versións de CUDA.

Non obstante, a métrica que máis nos interesa é o tempo de execución de cada algoritmo. Este valor utilizámolo despois para comparar o rendemento das distintas implementacións.

[2] nvprof - https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview
[3] NVIDIA Nsight Compute - https://developer.nvidia.com/nsight-compute

Sen empregar ningunha ferramenta adicional, temos dúas formas de rexistralo dende o interior do código:

- Para medir o tempo de execución das funcións síncronas, é dicir, aquelas que bloquean a execución da CPU ata a súa finalización, podemos usar unha libraría como a C Time Library[4], que permite calcular o tempo transcorrido entre dous momentos temporais, rexistrados mediante a chamada á función `clock()`. Nos distintos programas implementados, para rexistrar o tempo de execución en milisegundos da función `cpuConvolution`, utilizouse a seguinte fórmula:

```
clock_t t1 = clock();
cpuFunction( ... );
float milliseconds = (float)(clock() - t1) / CLOCKS_PER_SEC * 1000;
```

- Para medir o tempo de execución dos *kernels* CUDA podemos empregar o método anterior, sempre que fagamos a sincronización do dispositivo (`cudaDeviceSynchronize`) xusto antes de chamar outra vez á función `clock()`. Non obstante, CUDA proporciona un método para levar a cabo esta acción de forma máis precisa, os CUDA Events[5]. O procedemento sería o seguinte:

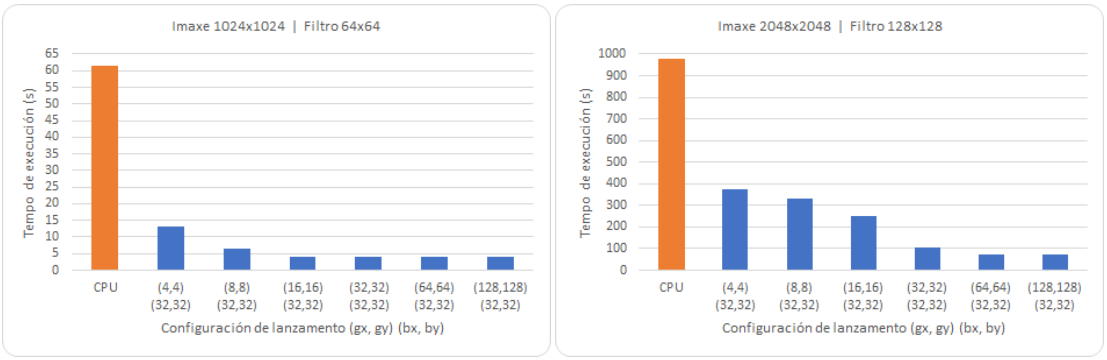
```
// Inicialización dos eventos
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

// Rexistro dos eventos
cudaEventRecord(start);
gpuKernel <<< grid,block >>>( ... );
cudaEventRecord(stop);

// Cálculo do tempo transcorrido entre eventos
cudaEventSynchronize(stop);
cudaEventElapsedTime(&milliseconds, start, stop);

// Destrución dos eventos
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

[4]C Time Library - http://www.cplusplus.com/reference/ctime/
[5]CUDA's Event API - https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EVENT.html



(a) Imaxe de 3x1024x1024 e filtro de 3x64x64 elementos

(b) Imaxe de 2048x2048 e filtro de 128x128 elementos

Figura 5.1: Tempo de execución da convolución 2D en función da configuración de lanzamento (dimensións do *grid* e dimensións dos bloques) para dous conxuntos de datos de entrada.

En todas as probas realizadas, empregamos o primeiro método (*CPU Timers*) para obter o tempo de execución na CPU e o comando `nvprof`, ou Nsight Compute, para obter o tempo de execución na GPU.

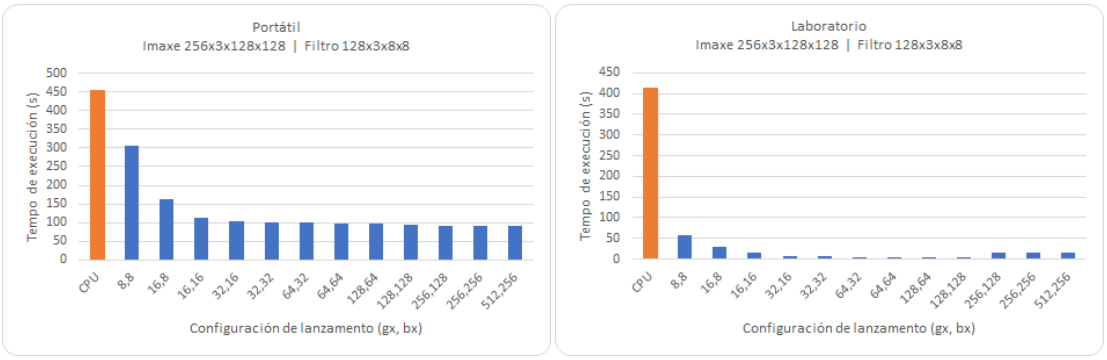
5.2 Análise dos resultados

Nesta sección estudaremos versión a versión os resultados obtidos con cada mellora introducida. Dita análise, realizada ao final de cada iteración, serviu para planificar o traballo e as melloras a desenvolver na seguinte iteración. Dedicaremos un apartado a cada unha das implementacións citadas no capítulo 4.

5.2.1 Primeira versión

Aínda que a versión v1.0 (capítulo 4, sección 4.1) non implementa a convolución por lotes, senón simplemente a convolución 2D entre unha imaxe e un filtro, xa podemos prever polos resultados obtidos da súa execución que a paralelización do procesamento na GPU vai supor un factor de mellora moi importante.

Para ter unha idea de como afecta o nivel de concorrencia ao tempo de execución, realizamos varias probas fixando o tamaño dos datos e variando a configuración de lanzamento, é dicir, as dimensións do *grid* e do bloque (número de fíos lanzados). O conxunto de datos de maior tamaño está composto por unha imaxe de 2048x2048 elementos e un filtro de 128x128, ambos de tres canles. Na elección deste conxunto buscouse que tivese moitos elementos e, polo tanto, que ocupase gran parte da memoria da GPU, pero que á vez permitise executar o programa na CPU, é dicir, que o tempo de execución estivese dentro duns límites razoables.



(a) Equipo portátil (b) Equipo laboratorio

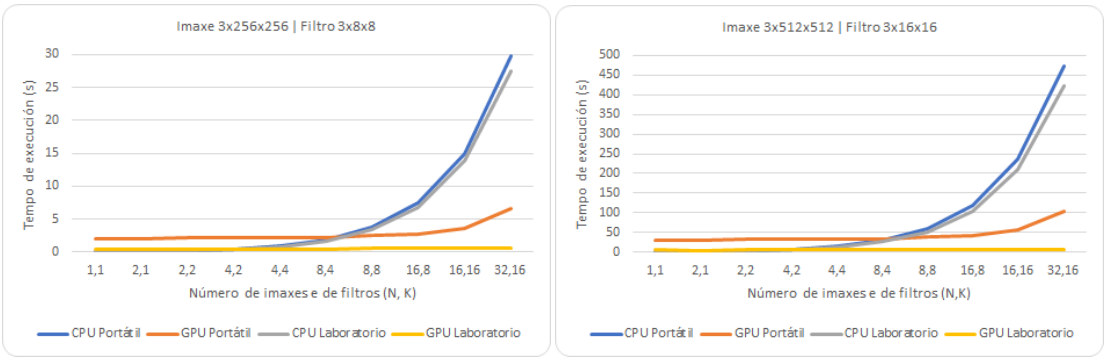
Figura 5.2: Tempo de execución do algoritmo directo en función da configuración de lanzamento, en ambos equipos.

Como podemos ver na figura 5.1, conforme aumenta o número de fíos o tempo de execución diminúe considerablemente. Non obstante, existe un límite ao paralelismo que podemos aproveitar, cando xa non é posible seguir dividindo o problema en anacos máis pequenos o tempo de execución deixa de baixar. Por exemplo, na figura 5.1 (b) podemos ver que para este algoritmo a configuración de maior tamaño que podemos utilizar é un *grid* formado por *64x64* bloques e bloques de *32x32* fíos, en total, unha malla de *2048x2048* fíos, as mesmas dimensións que para a imaxe de entrada. Se comparamos ambas gráficas decatámonos de que, cun maior tamaño de entrada e poucos bloques, a diferencia entre a GPU e a CPU é menor, xa que desta forma non se aproveita todo o paralelismo e a latencia da GPU adoita ser sempre menor. En termos de aceleración, comprobamos de maneira empírica que a execución na GPU pode chegar a ser, dependendo dos datos de entrada, ata 16x veces máis rápida que a versión na CPU. Finalmente, cabe destacar que todas as probas deseñadas para esta versión foron executadas na tarxeta NVIDIA GeForce 840M do equipo portátil.

5.2.2 Segunda versión

Na versión v2.0 (capítulo 4, sección 4.2), cambia o algoritmo e a forma de paralelizalo. Pasamos dun filtrado de imaxes no dominio do espazo, mediante a convolución 2D dunha imaxe e un filtro, á operación que realizan as redes neuronais convolucionais (a convolución directa) cun lote de imaxes e varios filtros. Ademais, nesta ocasión utilizamos un *grid* dunha única dimensión e paralelizamos o procesamento das imaxes e os filtros, pero non dos seus elementos. Agora veremos como afectan estas decisións ao rendemento do programa.

En primeiro lugar probamos a fixar, igual que na versión anterior, as dimensións dos datos de entrada e cambiamos a configuración de lanzamento. Na figura 5.2 (a) representamos os



(a) Imaxe de 256x256 e filtro de 8x8 elementos, 3 mapas de características.

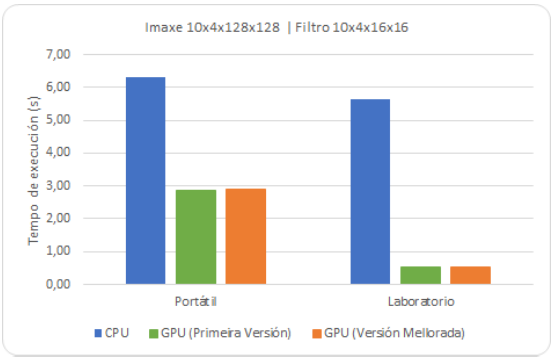
(b) Imaxe de 512x512 e filtro de 16x16 elementos, 3 mapas de características.

Figura 5.3: Tempo de execución do algoritmo directo en función do tamaño dos datos, nas distintas plataformas.

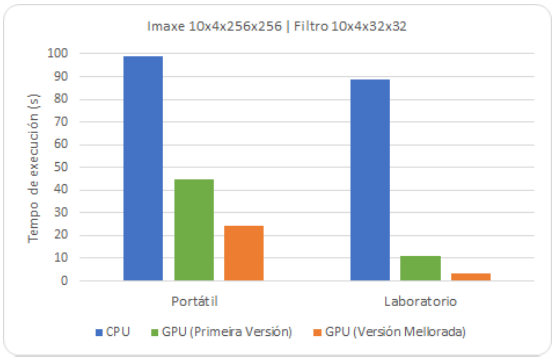
resultados obtidos para unha entrada de dimensións *256x3x128x128* e un filtro de *128x3x8x8*. Repetimos agora as probas no equipo do laboratorio, figura 5.2 (b), que posúe unha GPU de gama media-alta. A primeira impresión tras esta primeira proba é que, comparando os resultados da gama baixa, a diferencia temporal entre a GPU e a CPU non é tan evidente como o era na anterior versión.

En segundo lugar, fixamos a configuración de lanzamento no valor óptimo para este *kernel* e executamos o programa cambiando as dimensións dos datos de entrada. Máis concretamente, imos aumentando o tamaño de *batch* e o número de filtros, as únicas variables paralelizadas nesta función. Para rexistrar os tempos de execución representados na figura 5.3, partimos dunha entrada cunha imaxe e un filtro, cuxo rendemento é claramente inferior ao da CPU; mais en canto comezamos a aumentar ditos valores a diferencia vólvese cada vez menor, chegando a superala con gran vantaxe. Isto débese a que o tempo de execución na CPU aumenta na mesma proporción que o tamaño da entrada, mentres que o tempo de execución da GPU mantense bastante constante debido á paralelización do procesamento.

En conclusión, unha vez feitas distintas probas con distinto número de datos, decatámonos de que o nivel de paralelismo aproveitado nesta versión non é suficiente. Aínda que o rendemento xeral do programa CUDA supere ao secuencial, o tempo de execución do programa con conxuntos de datos de gran tamaño, como os empregados nas aplicacións de intelixencia artificial é excesivo. Como explicábamos no capítulo correspondente, neste algoritmo só paralelizamos os dous bucles máis externos da función (o do tamaño de *batch* e o do número de filtros) polo que desaproveitamos a maior parte do paralelismo. As posteriores versións corrixirán este problema.



(a) 10 imaxes de 128x128 e 10 filtros de 16x16 elementos, 4 mapas de características.



(b) 10 imaxes de 256x256 e 10 filtros de 32x32 elementos, 4 mapas de características.

Figura 5.4: Tempo de execución da versión mellorada do algoritmo directo en ambos equipos, confrontándoo cos resultados previos.

5.2.3 Terceira versión

As últimas probas realizadas correspóndense coa versión v3.3 deste proxecto (capítulo 4 sección 4.3). Como xa explicamos, esta versión substitúe a implementación directa da convolución por unha baseada no algoritmo GEMM. Nesta ocasión, tendo en conta que o novo código busca mellorar a versión anterior, as probas foron realizadas tomando como versión de referencia o algoritmo directo da convolución CNN, é dicir, o que explicamos no capítulo 4 sección 4.2. Para que a comparación fose máis xusta, solucionamos o problema da falta de paralelismo que detectamos no apartado anterior, dividindo o procesamento de todos os bucles do algoritmo entre os distintos fíos, a diferencia da versión anterior na que só dividíamos os dous primeiros. En total, imos comparar seis funcións, as catro implementadas nesta versión, a de referencia e tamén o algoritmo GEMM que implementa a libraría cuDNN. É necesario precisar que mediante esta implementación non pretendemos acadar o rendemento da libraría cuDNN, unha libraría desenvolvida por un equipo de enxeñeiros da propia NVIDIA, pero si é interesante observar os seus resultados en contraposición aos nosos.

Antes de comezar a análise dos novos algoritmos, aumentamos o aproveitamento do paralelismo na versión de referencia e cuantificamos a mellora. Tras realizar varias probas con distinto número de datos chegamos á conclusión de que a versión mellorada, comparando ambas coa mellor configuración de lanzamento en cada caso, mellora de forma xeneralizada o rendemento da súa predecesora, sendo naqueles casos onde os tamaño das imaxes e dos filtros é maior nos que se aprecia un maior incremento. Na máquina do laboratorio os resultados son similares, aínda que a diferencia coa CPU é incluso maior. Na figura 5.4 podemos ver unha comparativa en ambos equipos do tempo de execución dos programas, empregando

dous conxuntos de datos distintos, un co tamaño de imaxe e de filtro máis grande que podemos procesar (debido ás limitacións de memoria da propia GPU) e outro que divide á metade a altura e a largura das imaxes e dos filtros do anterior, co obxectivo de recalcar a diferenza entre resultados. Observamos que nos resultados do conxunto máis pequeno non se aprecia diferencia algunha, en cambio, como a nova versión é capaz de acadar un maior nivel de paralelismo, esta última é moito máis axeitada cando as entradas á función son de gran tamaño.

A continuación probamos os distintos algoritmos empregando para todos eles a mesma configuración de lanzamento. A configuración escollida para a primeira aproximación (*Gemm*) é de N bloques (tantos coma imaxes no *batch*) e 1024 fíos, xa que no interior do *kernel* repartimos, debido as necesidades do algoritmo, as imaxes do *batch* entre os distintos bloques. Na segunda aproximación (*Split*), para os tres conxuntos de filtros foron lanzados 24 bloques de 1024 fíos cada un. Respecto ao conxunto de datos, definimos dúas imaxes e tres filtros, ademais nas probas imos incrementando o número de imaxes no lote de 1 a 10. Na figura 5.5 representamos graficamente o resultado para unha imaxe de *3x128x128* elementos e na figura 5.6 ampliamos o número de elementos a *3x256x256*. Debido ao crecente espazo de traballo que require o algoritmo GEMM, na figura 5.6 non puidemos utilizar os filtros de maior tamaño (64x64). Nestas figuras observamos que, en xeral, conforme aumenta o número de imaxes no *batch* o tempo de execución tamén sobe, como é de esperar. Non obstante tamén detectamos que, para filtros de 16x16 en adiante, o rendemento da versión GEMM é inferior en case todos os casos á versión de referencia, de feito, a nova versión unicamente supera á anterior nos casos onde hai moitas imaxes e filtros, e o tamaño destes últimos é pequeno. Un exemplo disto son as figuras 5.6b e 5.7. O escaso rendemento desta versión é debido, por un lado, á forma de dividir o traballo entre os bloques que, como comentamos no capítulo anterior, pode infrautilizar as capacidades da GPU. Outro factor a ter en conta é a sobrecarga da barreira de sincronización utilizada entre as funcións que conforman o algoritmo. En definitiva, para compensar o traballo a maiores que realiza este algoritmo, deberíamos implementar algunha optimización sobre a multiplicación de matrices, como pode ser a vectorización ou operacións como *tiling* e *unrolling* sobre os bucles [24]. Para comprobar esta hipótese, substituímos o *kernel* do produto de matrices na versión *Split* por unha función equivalente da libraría cuBLAS (ver capítulo 4 apartado 4.3.3) e representamos os resultados desta versión, que identificamos cunha liña amarela, xunto ás gráficas da versión *Split*. Nestes resultados si que observamos unha mellora evidente do rendemento fronte á versión directa do algoritmo.

Por outro lado, é de esperar que a versión do algoritmo GEMM que emprega *Shared Memory* supera o rendemento da que non o usa, aínda que a diferencia sexa pequena. Isto non ocorre en cambio na segunda aproximación (chamada *Split* na figura), onde é necesario

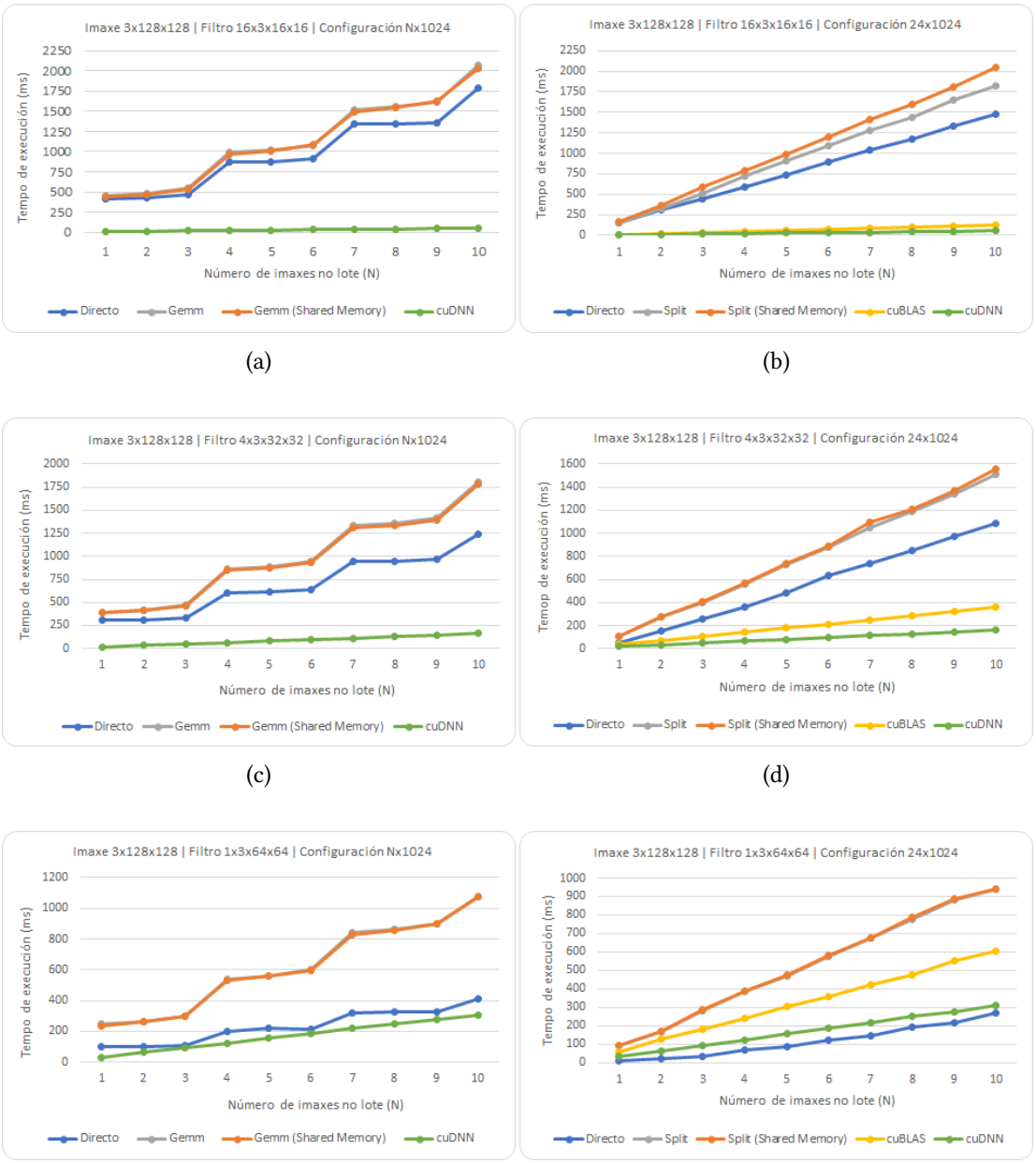


Figura 5.5: Resultados da execución da primeira aproximación (*Gemm*, á esquerda) e da segunda (*Split*, á dereita), cunha imaxe de 3x128x128 e tres filtros diferentes.

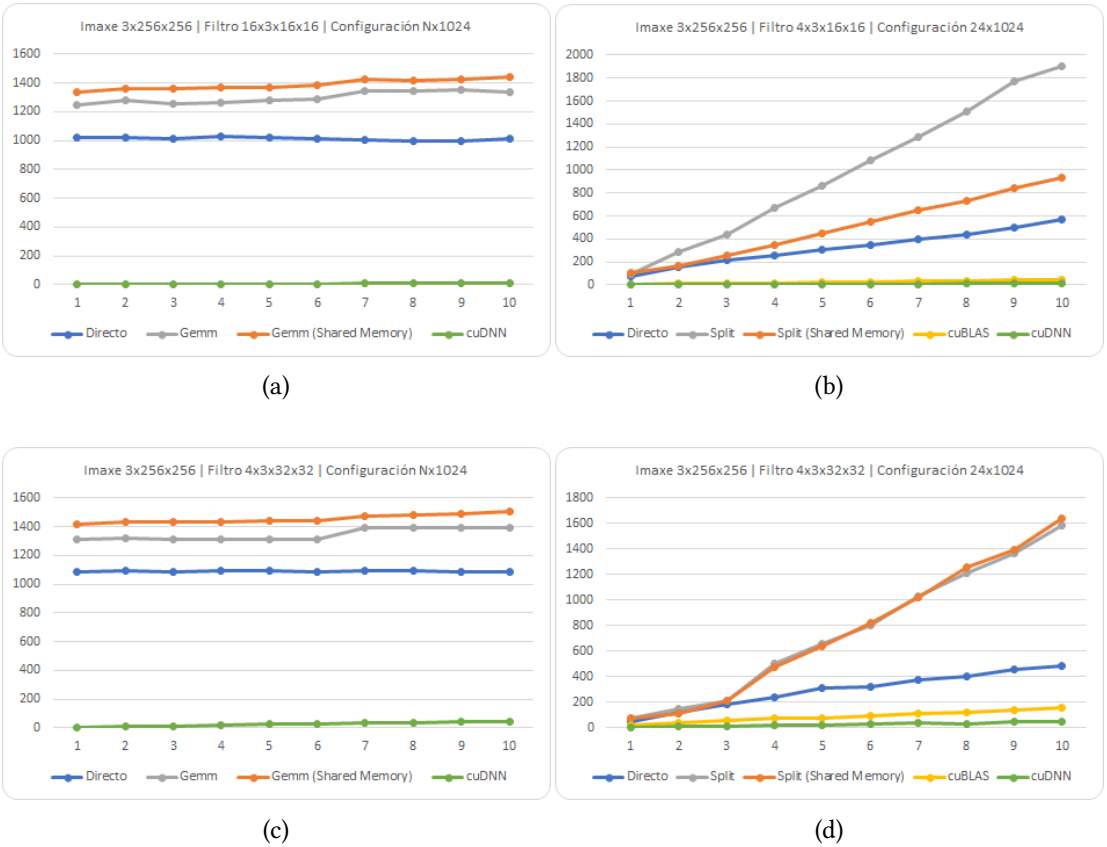


Figura 5.6: Resultados da execución da primeira aproximación (*Gemm*, á esquerda) e da segunda (*Split*, á dereita), cunha imaxe de 3x256x256 e dous filtros diferentes.

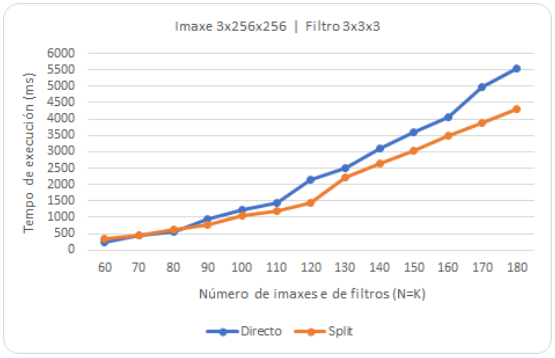


Figura 5.7: Tempo de execución do algoritmo GEMM (versión *Split*) cun filtro de pequeno tamaño e grandes conxuntos de datos, supera o rendemento da versión directa.

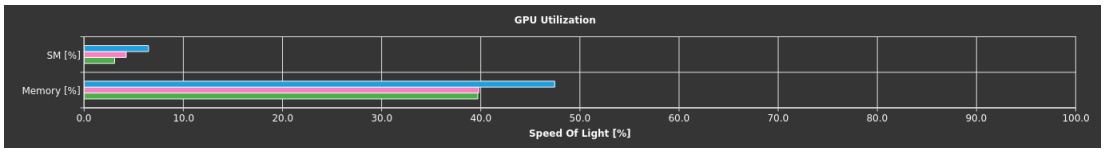
invertir a orde dos dous bucles máis externos e, polo tanto, os accesos a memoria deixan de ser secuenciais.

Para rematar esta análise, incluímos na figura 5.5 o resultado do algoritmo GEMM realizado pola libraría cuDNN que, se nos fixamos nas gráficas da figura, é practicamente inapreciable ao lado do resto, excepto nun caso. Na última figura podemos ver como afecta a configuración de lanzamento ao algoritmo da convolución directa. Cando lanzábamos un número de bloques igual ao tamaño de *batch*, o tempo de execución superaba sempre á versión cuDNN, mentres que cando aumentamos o número de bloques a 24, a nosa aproximación é máis rápida. Por outro lado, nesa proba utilizouse un único filtro de 3x64x64 (non caben máis dese tamaño na *Shared Memory*) e cuDNN está optimizada para procesar lotes de moitas imaxes e filtros. Hai que ter en conta tamén, que estamos comparando o algoritmo directo co algoritmo GEMM, xa que cuDNN aínda non ten implementado o algoritmo da convolución directa.

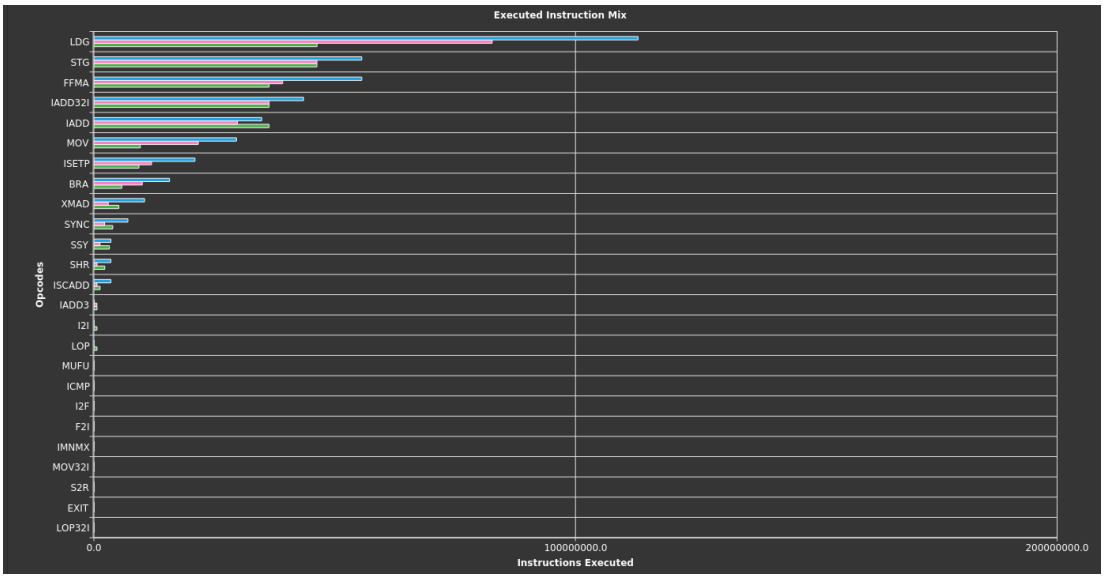
Análise con Nsight Compute

Finalmente, para afondar nas causas dos resultados obtidos, executamos os algoritmos baixo o *profiler* Nsight Compute. Con esta ferramenta somos capaces de visualizar de forma sinxela parámetros como a utilización da GPU e dos distintos tipos de memoria, a ocupación dos bloques e dos *warps*, ou o tempo que estes pasan esperando polos distintos recursos. Debido á relevancia que teñen os *warps* na análise realizada por Nsight Compute, é necesario recordar este concepto (explicado no capítulo 2 apartado 2.2.2), consistente nunha agrupación de 32 fíos a nivel hardware, que comparten certos recursos do multiprocesador. Para comezar esta análise, identificamos as causas máis habituais dun rendemento precario, por exemplo a baixa latencia dos accesos a memoria ou a escasa ocupación dos bloques e os *warps*, e despois buscamos indicios disto na análise realizada por Nsight Compute. A continuación comentamos os resultados obtidos tras o perfilado das distintas versións do algoritmo.

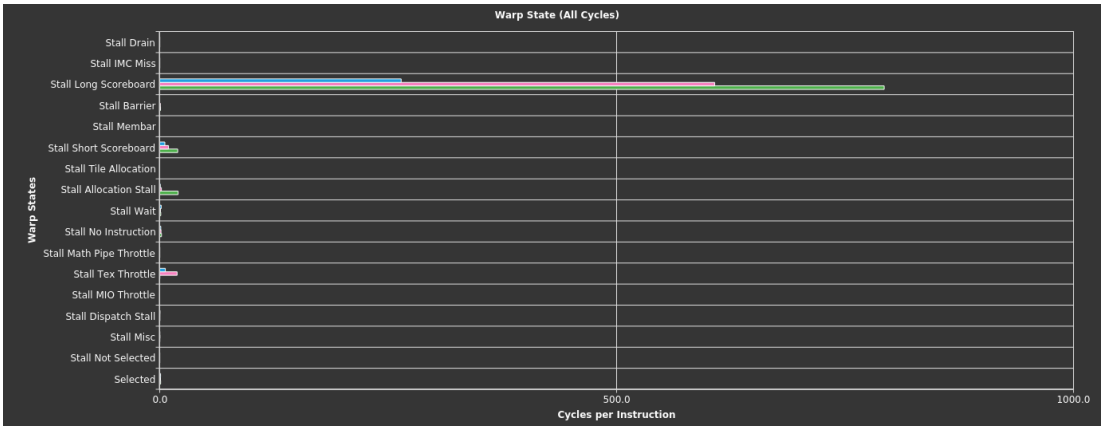
Por un lado podemos ver de forma gráfica que, aínda que a versión directa ocupa máis tempo de GPU (figura 5.8a) e executa máis instrucións que o algoritmo GEMM (figura 5.8b), nesta outra os *warps* pasan máis tempo inactivos agardando pola súa quenda de acceso á memoria global (figura 5.8c), o que se traduce nunha maior duración do programa. Nas figuras anteriormente citadas adxuntamos algunhas das gráficas xeradas por Nsight Compute, onde destaca a cantidade de ciclos desaproveitados (métrica *Stall Long Scoreboard* que recolle a subfigura 5.8c) polos *warps* que agardan acceder á memoria global ou local, sobre todo no algoritmo GEMM que emprega *Shared Memory* (en verde). O resto de parámetros representan outro tipo de esperas pero, como podemos ver na figura, non supoñen un problema de tanto calibre.



(a) Tempo de ocupación da GPU e da memoria.



(b) Número de instrucións executadas, en función do tipo.



(c) Tempo de agarda (*stall*) por distintos motivos.

Figura 5.8: Métricas recollidas por Nsight Compute dos algoritmos directo (en azul), GEMM (en laranxa) e GEMM empregando *Shared Memory* (en verde).

Por outro lado, como adiantábamos ao principio deste apartado, é interesante coñecer como afecta a configuración de lanzamento ao nivel de ocupación [13] dos *warps*. En función de distintos parámetros da propia tarxeta, como a *CUDA Compute Capability*, ou do *kernel*, como o número de rexistros empregados ou os fíos lanzados, podemos calcular o nivel de ocupación (aproveitamento) da GPU. O cálculo pódese facer a través dunha calculadora dispoñible na documentación de CUDA ou ben delegando este traballo en Nsight Compute. As gráficas de ocupación da versión directa (en azul), da segunda aproximación (*Split*, en morado) e da súa versión con *Shared Memory* (en laranxa) móstranse na figura 5.9. Na segunda gráfica desta figura podemos observar facilmente que a mellor configuración de lanzamento para cada un dos *kernels* citados é empregar 576, 768 e 640 fíos por bloque, respectivamente, cando o límite está en 1024 fíos por bloque. Pero a configuración de lanzamento do *kernel* non é o único parámetro que inflúe na ocupación da GPU, aínda que si o máis sinxelo de modificar. Outro dos parámetros que pode afectar é o número de rexistros que usa cada fío, na primeira gráfica da figura 5.9 vemos como varía a ocupación en función deste valor. Así mesmo, tamén inflúe o uso da *Shared Memory* que realicen os bloques, podemos velo na terceira gráfica da figura 5.9. En definitiva, dos datos aportados nesta figura sacamos a conclusión de que, a diferencia do que pode parecer a priori, aumentar demasiado o número de fíos por bloque pode ser contraproducente. É por iso que debemos ter coidado naqueles casos onde teñamos un gran número de datos e queiramos aumentar o nivel de concorrencia aproveitado. O ideal é procurar manter un alto nivel de ocupación, por exemplo fixando o número de fíos por bloque no valor recomendado, e aumentar o tamaño do *grid* cando o algoritmo o permita.

Finalmente, buscamos na análise realizada por Nsight Compute os datos de utilización da memoria compartida, co obxectivo de afondar nos detalles da pequena mellora introducida pola súa utilización. Concretamente, obtivemos o número de instrucións de lectura/escritura que executan os distintos *kernels* sobre esta memoria. Na figura 5.10 adxuntamos un esquema moi interesante onde se representan todos os tipos de memoria e o número total de bytes transmitidos (ou o número de transaccións) a través dos buses de comunicación. A figura foi obtida a partir dos resultados da execución do *kernel* `matrixProduct_shared` cun filtro de tamaño *4x3x32x32*. A elección deste filtro débese a que o seu tamaño equivale á capacidade da propia memoria (49152 bytes), desta forma resaltamos o nivel de utilización deste tipo de memoria. En total, nesta execución foron realizadas 2304 operacións de escritura e 37748736 operacións de lectura, é dicir, dezaseismil veces máis operacións de lectura. Este é o tipo de situación nas que a *Shared Memory* pode ser de gran utilidade, non obstante, como puidemos comprobar en anteriores probas e para este *kernel* en concreto, a súa vantaxe fica enmascarada polo sobrecusto que supón o cambio do patrón de acceso a memoria.

Figura 5.9: Nivel de ocupación dos multiprocesadores da GPU para distintas configuracións de lanzamento nos *kernels* da convolución directa (en azul), da segunda aproximación do algoritmo GEMM (*Split*, en morado) e da súa versión con *Shared Memory* (en laranxa).

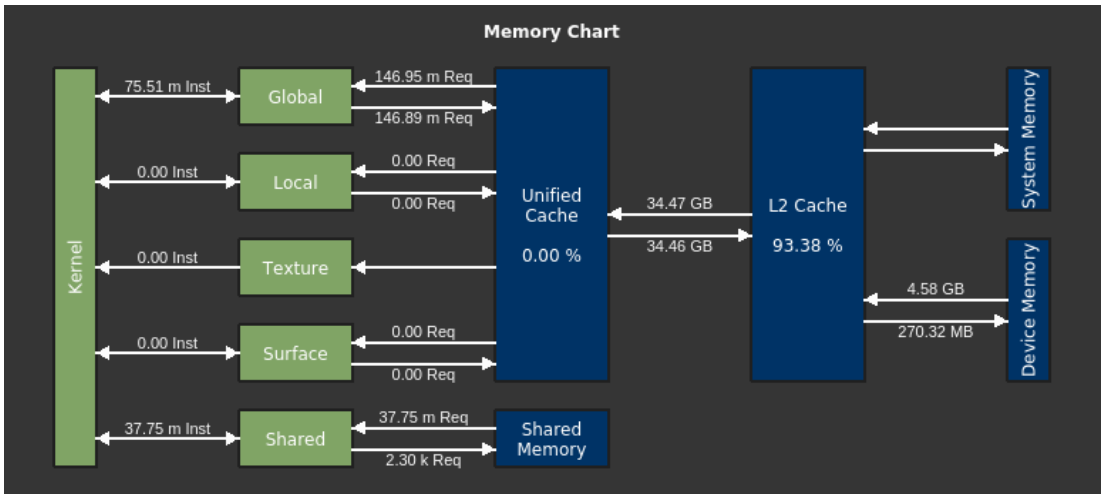


Figura 5.10: Transferencias de datos entre as distintas memorias durante a execución do *kernel* `matrixProduct_shared`, correspondente á versión con *Shared Memory* da segunda aproximación (*Split*) do algoritmo GEMM.

Capítulo 6

Conclusións

No derradeiro capítulo desta memoria, realizaremos un breve repaso do traballo desenvolvido no proxecto. Resumiremos tamén a situación final do mesmo, comentando a súa concordancia cos obxectivos definidos inicialmente.

Desta forma, na sección 6.1 realizarase o citado repaso e o contraste de obxectivos. A continuación, na sección 6.2 mencionarase a relación entre o coñecemento adquirido durante a realización do proxecto e as competencias da titulación. Finalmente, na sección 6.3, comentaranse as posibles liñas de traballo futuras, así como as leccións aprendidas.

6.1 Recapitulación

Neste proxecto implementáronse varios métodos para realizar a operación da convolución en lotes, sendo este o obxectivo principal do traballo. Para iso, tal e como contemplamos nos obxectivos citados no capítulo 1, tivemos que estudar e comprender como funciona esta operación e as súas aplicacións no campo das redes de aprendizaxe profunda, recollendo este coñecemento no capítulo 2. Ademais, propuxemos múltiples variantes para a realización da convolución e implementámolas nun proceso iterativo que plasmamos no capítulo 4. Finalmente, no capítulo 5 analizamos e comparamos o rendemento das distintas propostas. Podemos afirmar, polo tanto, que foron cubertos todos os obxectivos inicialmente definidos para este proxecto.

O resultado de todo este traballo de estudo e implementación é, por un lado, un conxunto de algoritmos que realizan de distintas formas a convolución, e unha análise do rendemento de cada un que nos proporciona a información necesaria para seguir mellorándoos. Por outro lado, a nivel persoal, deste traballo obtiven unha boa base de coñecemento sobre a programación de propósito xeral en GPUs e, concretamente, sobre a linguaxe de programación CUDA.

Este campo, ademais, está a ser de gran interese na resolución de problemas en distintos ámbitos, debido a evidente vantaxe que presentan as GPUs cando se trata de procesar unha mesma operación sobre múltiples datos.

6.2 Competencias da titulación

Houbo varias materias do Grao en Enxeñaría Informática nas que estudamos moitos dos conceptos aplicados neste traballo. Partindo da base, a materia *Concorrencia e Paralelismo* foi o noso primeiro contacto coa programación paralela. Máis adiante, nas materias propias da mención en Enxeñaría dos Computadores, como *Arquitectura de Computadores*, afianzamos o concepto de paralelismo e empregamos, por exemplo, extensións SSE (do inglés, *Streaming SIMD Extensions*) que tamén aproveitan as características SIMD dos dispositivos. Por outro lado, en *Procesamento Dixital da Información* estudamos o concepto de convolución e realizamos o filtrado, tanto espacial como en frecuencia, de imaxes. Por suposto, todo isto non sería posible se antes non tivésemos asimilado os conceptos básicos sobre a programación, en materias como *Programación I* e *Programación II*, ou sobre a arquitectura dos computadores, en *Fundamentos* e *Estrutura de Computadores*, onde aprendimos conceptos básicos sobre os compoñentes, a estrutura e o funcionamento da memoria e os procesadores.

Tamén é posible relacionar o traballo realizado cun gran número de competencias propias do título [25], das que destacan as seguintes:

- A12: *Coñecemento e aplicación dos procedementos algorítmicos básicos das tecnoloxías informáticas para deseñar solucións a problemas, analizando a idoneidade e a complexidade dos algoritmos propostos.*

  Competencia que define os propios obxectivos do proxecto, xa que nel desenvolvemos distintos algoritmos que resolven un mesmo problema e valoramos o seu rendemento.

- A15: *Capacidade de coñecer, comprender e avaliar a estrutura e a arquitectura dos computadores, así como os compoñentes básicos que os conforman.*

  Co obxectivo de analizar a idoneidade das tarxetas gráficas na resolución de determinados problemas, realizamos un estudo da arquitectura e compoñentes destes dispositivos.

- A20: *Coñecemento e aplicación dos principios fundamentais e técnicas básicas da programación paralela, concorrente, distribuída e de tempo real.*

  As optimizacións levadas a cabo nos algoritmos implementados están baseadas principalmente na paralelización do procesamento en múltiples fíos.

- A41: *Capacidade para avaliar a complexidade computacional dun problema, coñecer estratexias algorítmicas que poidan conducir á súa resolución e recomendar, desenvolver e implementar aquela que garanta o mellor rendemento de acordo cos requisitos establecidos.*

  Antes de comezar calquera das iteracións levadas a cabo neste proxecto, foi necesaria unha análise previa do problema a resolver en dita iteración, un estudo das posibles solucións e finalmente, a selección e implementación dunha das variantes propostas.

6.3 Liñas futuras

Tras analizar o rendemento dos algoritmos, en especial do algoritmo GEMM, decatámonos de que é posible introducir certas optimizacións na multiplicación de matrices. Esta operación é unha gran coñecida e leva sendo estudada moitos anos, polo que poderíamos aproveitar todo o coñecemento do que dispoñemos sobre ela para acelerar o procesamento global da convolución.

Algunhas das melloras que poderíamos introducir baséanse, por un lado, na optimización dos patróns de acceso á memoria, buscando a configuración que mellor favoreza a permanencia dos datos na caché e evitando así os fallos caché que provocan unha maior latencia na resposta. Neste sentido, existen técnicas como o *loop tiling*, que consiste en dividir as iteracións dun bucle en bloques de menor tamaño, o que conduce á partición dos datos empregados en dita iteración axustándoos ao tamaño da caché. Desta forma nos aseguramos de que os datos permanecen na caché ata a súa reutilización. Por outro lado, a coñecida técnica do *loop unrolling* permite reducir o traballo que realiza o procesador calculando os saltos dos bucles. Por exemplo, poderíamos transformar manualmente oito iteracións dunha única instrución en dous iteracións de catro instrucións cada unha delas.

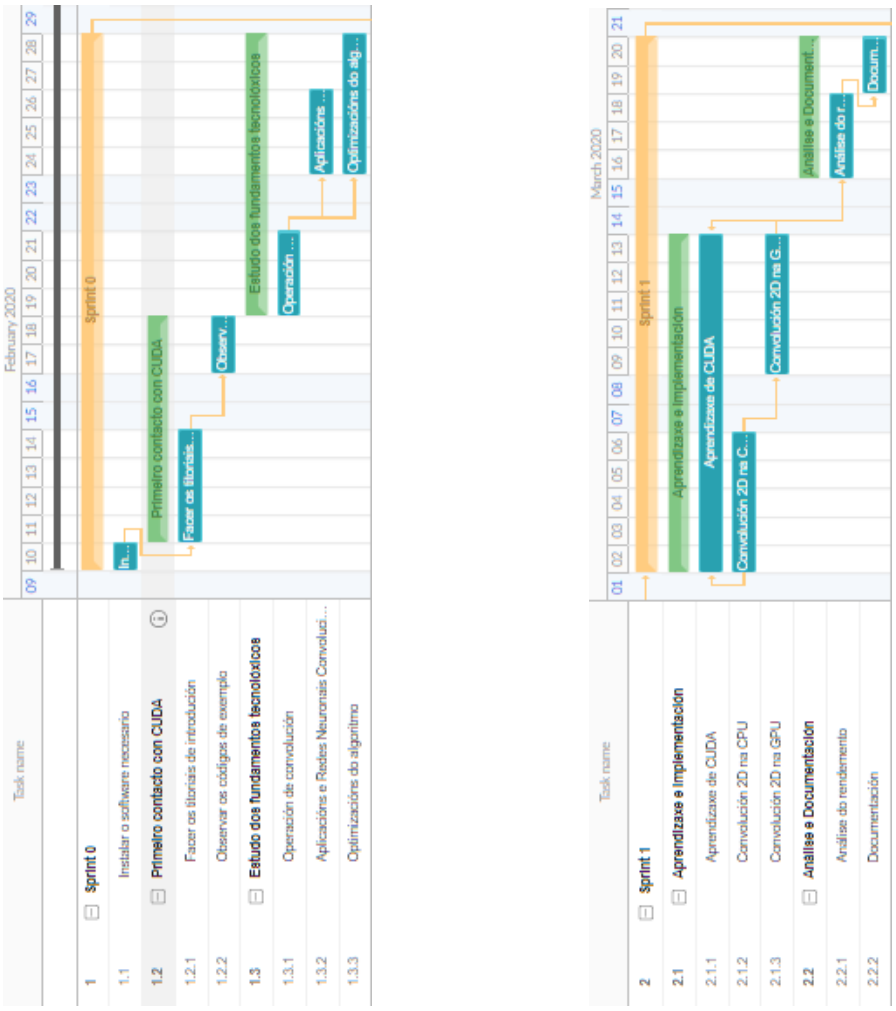
Finalmente, habería que estudar a posibilidade de reducir a latencia dos accesos á memoria favorecendo a coalescencia das operacións de lectura, é dicir, unindo varias destas operacións nunha única transferencia. Para tentar esta optimización, aplicable sobre calquera dos algoritmos desenvolvidos ata agora, podemos probar diferentes técnicas. Unha relativamente sinxela consiste en aliñar as estruturas de datos utilizadas á hora de definilas, completando incluso o seu tamaño cando non cumprise cos requisitos de aliñamento. Outra posible mellora do rendemento podería consistir na transformación previa dos datos antes de ser operados, por exemplo, poderíamos alterar a disposición en memoria das imaxes e dos filtros e preparar os algoritmos para procesar imaxes nos formatos NHWC ou NC/xHWx, explicados no capítulo 4, en lugar do NCHW que empregamos ata agora. Por outro lado, ademais de traballar sobre o patrón de acceso á memoria, poderíamos cambiar o tipo de dato empregado para al-

macenar a información das entradas e as saídas dos *kernels* e utilizar un dos que proporciona a propia API de CUDA, que ademais de estar aliñados poden aproveitar as capacidades de vectorización da tarxeta.

Apéndices

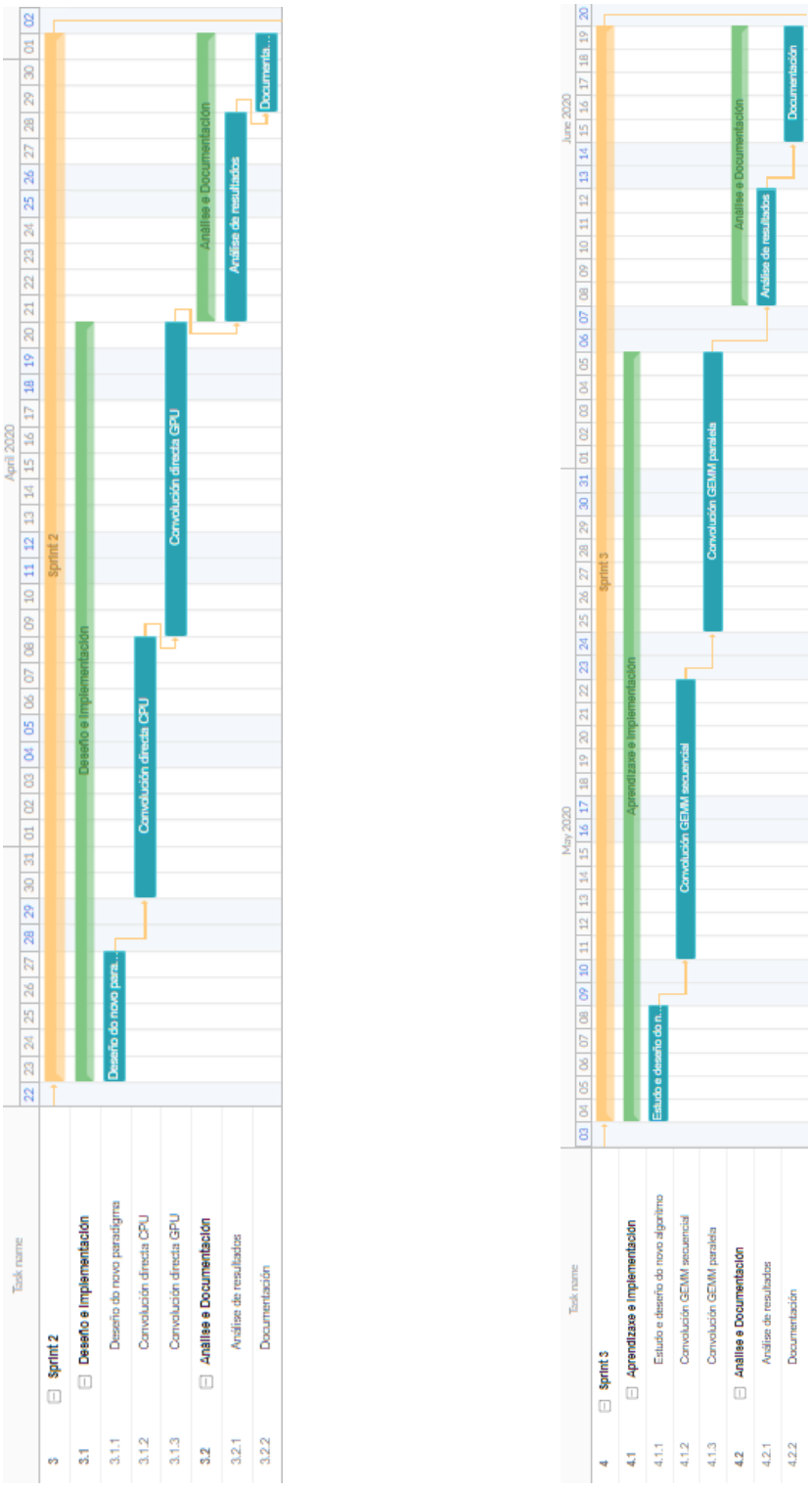
Apéndice A

Diagrama de Gantt



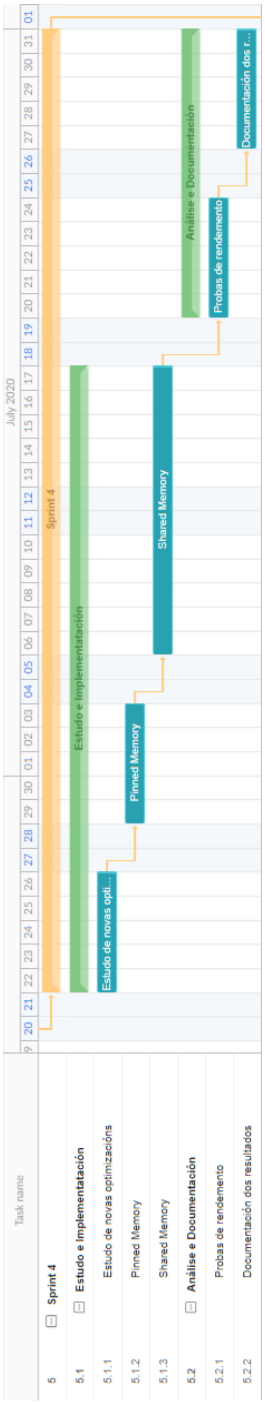
(a) Sprint 0 (b) Sprint 1

Figura A.1: Diagrama de Gantt dos Sprints 0 e 1

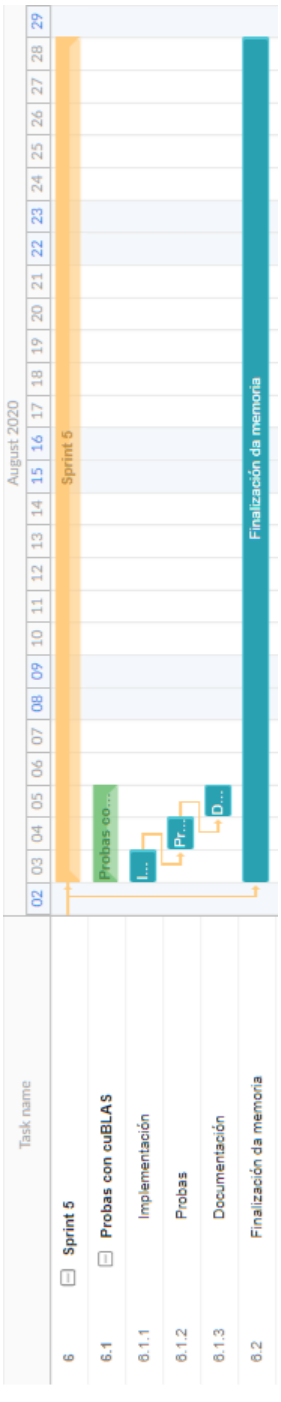


(a) Sprint 2 (b) Sprint 3

Figura A.2: Diagrama de Gantt dos Sprints 2 e 3



(a) Sprint 4



(b) Sprint 5

Figura A.3: Diagrama de Gantt dos Sprints 4 e 5

Relación de Acrónimos

ALU *Arithmetic Logic Unit.*

ANN *Artificial Neural Network.*

API *Application Programming Interface.*

BLAS *Basic Linear Algebra Subprograms.*

CNN *Convolutional Neural Network.*

CPU *Central Processing Unit.*

CUDA *Compute Unified Device Architecture.*

DBN *Deep Belief Network.*

DNN *Deep Neural Network.*

DRAM *Device Random Access Memory.*

FFT *Fast Fourier Transform.*

FIR *Finite Impulse Response.*

FPGA *Field Programmable Gate Array.*

GAN *Generative Adversarial Network.*

GEMM *General Matrix Multiplication.*

GPGPU *General-Purpose Graphics Processing Unit.*

GPU *Graphics Processing Unit.*

HPC *High-Performance Computing.*

LOG *Laplacian Of Gaussian.*

SIMD *Single Instruction Multiple Data.*

SIMT *Single Instruction Multiple Data.*

SM *Streaming Multiprocessor.*

RBM *Restricted Boltzmann Machines.*

RNN *Recurrent Neural Network.*

UPN *Unsupervised Pretrained Network.*

Glosario

Autoencoder Rede neuronal capaz de aprender a codificar un conxunto de datos para despois reconstruir os datos orixinais a partir da súa representación codificada.

Batch Lote ou conxunto de imaxes que se utiliza como entrada á función da convolución.

Bias Parámetro das redes neuronais que se engade como unha constante ás entradas da rede.

Buffer Espazo de memoria utilizado para almacenar datos de forma temporal mentres estes son trasladados dun lugar a outro.

Compute Capability Valor que determina o conxunto de funcionalidades relacionadas coa capacidade de cómputo dunha GPU compatible con CUDA.

Convolución Operación matemática definida como a integral do produto entre dúas funcións, f e g, tras desprazar unha delas unha distancia t.

Cluster Sistema formado por varios computadores unidos entre si por unha rede de alta velocidade e coordinados para funcionar coma un único equipo.

Deep Learning Campo da intelixencia artificial no que se inclúen aquelas aplicacións baseadas en redes de aprendizaxe profunda, é dicir, aquelas que dispoñen dun maior número de capas ocultas.

Dilation Dilatación, parámetro da convolución que indica a separación entre elementos do filtro.

Framework (do inglés, marco de traballo) conxunto de ferramentas, ben sexan programas ou código, que serve de base para o desenvolvemento de software.

Grid Disposición lóxica dos fíos en execución nunha GPU compatible con CUDA. Ten dúas dimensións: número de bloques e número de fíos por bloque.

Host Sistema anfitrión, incluíndo CPU e memoria, en contraposición ao dispositivo invitado, neste caso a GPU.

Kernel Función CUDA que se executa na GPU, para o seu lanzamento é necesario indicar as dimensións do *grid*.

Loop Tiling Optimización que consiste en dividir as iteracións dun bucle en bloques de menor tamaño, o que conduce á partición dos datos empregados en dita iteración axustándoos ao tamaño da caché.

Loop Unrolling Optimización que permite reducir o traballo que realiza o procesador calculando os saltos dos bucles.

Macro Abreviatura de "macroinstrución", serie de instrucións almacenadas de forma que se poidan executar nunha única chamada.

Padding Marco que se engade ao redor dunha imaxe antes de utilizala como entrada na convolución, co obxectivo de evitar a redución que provoca dita operación no tamaño da saída.

Path Dirección composta por unha serie de nomes de directorio separados mediante un carácter (como pode ser "/" nos sistemas Unix) e que remata co nome dun ficheiro, de forma que indique a ruta do mesmo.

Pooling Operación realizada nas Redes Neuronais Convolucionais para reducir o tamaño dos datos e desta forma simplificar o procesado dos mesmos.

Profiling Técnica empregada para determinar as zonas máis conflitivas dun programa en termos de rendemento.

Sprint Iteracións nas que se basea a metodoloxía Scrum.

Stream Secuencia de operacións que se executan nunha GPU compatible con CUDA na orde indicada dende o *host*. Varios *streams* poden ser executados concorrentemente.

Stride Parámetro da convolución que indica o número de píxeles que debe saltar o filtro en cada iteración.

Warp Agrupación de 32 fíos a nivel hardware, que comparten recursos do multiprocesador.

Warp Scheduler Esquema que segue á GPU para planificar o comportamento dos fíos en cada *warp*.

Workspace Espazo de memoria utilizado para almacenar os datos que precisa o algoritmo para traballar.

Bibliografía

[1] P. Pröve, "An introduction to different types of convolutions in Deep Learning," *Towards Data Science*, 2017. Dispoñible en: https://towardsdatascience.com/types-of-convolutions-in-deep-learning-717013397f4d

[2] J. Patterson and A. Gibson, *Deep Learning*. O'Reilly Media, Inc., 2017. Dispoñible en: https://www.oreilly.com/library/view/deep-learning/9781491924570/ch04.html

[3] A. Karpathy *et al.*, "Convolutional Neural Networks for Visual Recognition, Course Notes," Standford University, Tech. Rep., 2020. Dispoñible en: http://cs231n.github.io/

[4] T. Gupta, "Deep Learning: Feedforward Neural Network," *Towards Data Science*, 2017. Dispoñible en: https://towardsdatascience.com/deep-learning-feedforward-neural-network-26a6705dbdc7

[5] W. Badr, "Auto-Encoder: What is it? And what is it used for?" *Towards Data Science*, 2019. Dispoñible en: https://towardsdatascience.com/auto-encoder-what-is-it-and-what-is-it-used-for-part-1-3e5c6f017726

[6] A. Sharma, "Restricted Boltzmann Machines — Simplified," *Towards Data Science*, 2018. Dispoñible en: https://towardsdatascience.com/restricted-boltzmann-machines-simplified-eab1e5878976

[7] M. Jordà, P. Valero-Lara, and A. J. Peña, "Performance Evaluation of cuDNN Convolution Algorithms on NVIDIA Volta GPUs," *IEEE Access*, vol. 7, pp. 70 461–70 473, 2019. Dispoñible en: https://ieeexplore.ieee.org/document/8721631

[8] E. Weisstein, "Convolution Theorem," MathWorld - A Wolfram Web Resource, Tech. Rep., 2020. Dispoñible en: https://mathworld.wolfram.com/ConvolutionTheorem.html

[9] S. Winograd, *Arithmetic complexity of computations*. SIAM: Society for Industrial and Applied Mathematics, 1980. Dispoñible en: https://books.google.es/books?id=wANiW8bGQpEC

[10] A. Lavin and S. Gray, "Fast Algorithms for Convolutional Neural Networks," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 4013–4021. Dispoñible en: https://ieeexplore.ieee.org/document/7780804

[11] NVIDIA, "CUDA C/C++ Programming Guide," 2020. Dispoñible en: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[12] M. Harris, "GPU Pro Tip: CUDA 7 Streams Simplify Concurrency," 2015. Dispoñible en: https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/

[13] NVIDIA, "Occupancy," in *CUDA C/C++ Best Practises Guide*. NVIDIA, 2020, ch. 10. Dispoñible en: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#occupancy

[14] P. Nee, "Introduction to GPGPU and CUDA Programming: Memory Coalescing," Center for Advanced Computing, Cornell University, Tech. Rep., 2013. Dispoñible en: https://cvw.cac.cornell.edu/GPU/coalesced

[15] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning," *ArXiv*, 2014. Dispoñible en: https://arxiv.org/abs/1410.0759

[16] oneAPI SRC, "oneAPI Deep Neural Network Library (oneDNN)." Dispoñible en: https://github.com/oneapi-src/oneDNN

[17] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," *arXiv preprint arXiv:1408.5093*, 2014. Dispoñible en: https://github.com/intel/caffe

[18] Google Brain Team, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015, software available from tensorflow.org. Dispoñible en: https://arxiv.org/abs/1603.04467

[19] NVIDIA, "cuBLAS API Reference," 2020. Dispoñible en: https://docs.nvidia.com/cuda/cublas/index.html

[20] Intel, "Intel® Math Kernel Library." Dispoñible en: https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html

[21] A. W. R. Fisher, S. Perkins and E. Wolfart., "Digital Filters," 2003. Dispoñible en: https://homepages.inf.ed.ac.uk/rbf/HIPR2/filtops.htm

[22] P. Jaumier, "Backpropagation in a convolutional layer," *Towards Data Science*, 2019. Dispoñible en: https://towardsdatascience.com/backpropagation-in-a-convolutional-layer-24c8d64d8509

[23] M. Harris, "Using Shared Memory in CUDA C/C++," 2013. Dispoñible en: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

[24] K. Goto and R. A. Van De Geijn, "Anatomy of High-Performance Matrix Multiplication," The University of Texas at Austin, Tech. Rep., 2006. Dispoñible en: https://www.cs.utexas.edu/~flame/pubs/GotoTOMS_revision.pdf

[25] "Facultade de Informática da Coruña. Guía docente: Competencias da titulación," 2019-2020. Dispoñible en: https://guiadocente.udc.es/guia_docent/index.php?centre=614&ensenyament=614G01&consulta=competencies&idioma=cat&any_academic=2019_20