Departamento de Enxeñaría de Computadores
Facultade de Informática da Coruña
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE MESTRADO
MESTRADO INTERUNIVERSITARIO EN
COMPUTACIÓN DE ALTAS PRESTACIÓNS

Implementación do algoritmo da convolución por lotes usando Intel oneAPI

Estudante: Sara Aguado Couselo
Dirección: Diego Andrade Canosa

A Coruña, xuño de 2021.

A miña nai

Agradecementos

Un saúdo aos meus compañeiros de mestrado, Bieito e Marco, polos bos ratos que pasamos na cafetaría virtual. Unha aperta tamén á miña compañeira e gran estudante Oluwatosin.

Resumo

Na actualidade, a operación de convolución está sendo obxecto de investigación debido á súa aplicación nas redes de aprendizaxe profunda. Esta operación alxébrica, ademais, é a que ocupa un maior tempo de cómputo no adestramento das redes neuronais convolucionais. Por outro lado, as plataformas heteroxéneas -aquelas que combinan máis dun tipo de procesador, por exemplo CPUs e GPUs- están tendo un gran auxe na resolución de problemas en diversos ámbitos, debido á súa capacidade de procesar de forma simultánea unha mesma operación sobre múltiples datos. Para aproveitar ao máximo esta característica e maximizar o número de datos a procesar, neste traballo decidiuse implementar o algoritmo da convolución por lotes. Utilizarase un *framework* de Intel, oneAPI, para traballar sobre plataformas heteroxéneas e probar o seu rendemento tanto en CPUs coma en GPUs. Ademais, realizarase o estudo do rendemento de distintas implementacións xa existentes.

Abstract

Nowadays, the convolution operation is being the subject of research due to its application in Deep Learning. This algebraic operation, moreover, is the one that occupies the most computational time in the training of convolutional neural networks. On the other hand, heterogeneous platforms -those that combine more than one type of processor, for example CPUs and GPUs- are having a great impact solving problems in several areas, due to their ability to simultaneously process the same operation on multiple data. To take advantage of this feature and maximize the number of data to be processed, in this work we decided to implement the batched convolution algorithm. An Intel framework, oneAPI, will be used to work on heterogeneous platforms and test its performance on both CPUs and GPUs. In addition, the performance of different existing implementations will be studied.

Palabras chave:

- Intel oneAPI
- Intel DevCloud
- Convolución por lotes
- Plataformas heteroxéneas
- Computación de altas prestacións
- GPGPU

Keywords:

- Intel oneAPI
- Intel DevCloud
- Batched convolution
- Heterogeneous platforms
- High Performance Computing
- GPGPU

Índice Xeral

Índice de Figuras

Índice de Táboas

Capítulo 1

Introdución

A optimización de rutinas recorrentes en aplicacións de gran valor computacional, é dicir, cun alto consumo de recursos de cómputo, é un interesante campo de estudo ou investigación. Nos últimos anos, asistimos á consolidación do *Deep Learning* como forma de atallar problemas de gran complexidade, e que requiren, precisamente, solucións de altas prestacións (HPC, *High Performance Computing*). Ao mesmo tempo, as plataformas heteroxéneas experimentaron un gran auxe na resolución de problemas ata entón reservados á CPU, debido principalmente á súa capacidade para responder á crecente demanda de datos das aplicacións. É por iso, que neste traballo decidiuse estudar unha das operacións máis habituais, e das que máis tempo consomen, dentro das redes de aprendizaxe profunda: a convolución, e ademais implementala nun *framework* compatible con sistemas que incorporen tanto CPUs como GPUs.

A continuación, na sección 1.1, veremos como pode axudar este proxecto a mellorar o rendemento do *Machine Learning* e que papel xogan nel as plataformas heteroxéneas. Máis adiante, na sección 1.2, comentaremos os principais obxectivos deste traballo de investigación. Finalmente, na sección 1.3, expoñeremos o resultado do proxecto, comprendendo como tal a ferramenta e a propia memoria.

1.1 Antecedentes

Por un lado, unha das aplicacións máis interesantes das redes de aprendizaxe profunda (DNN, *Deep Neural Networks*), ou máis concretamente, das redes neuronais convolucionais (CNN, *Convolutional Neural Networks*), consiste na análise e transformación de imaxes. Por desgraza, este tipo de aplicacións entrañan un alto consumo de recursos, tanto de almacenamento, pola gran cantidade e tamaño dos datos necesarios para o seu correcto funcionamento; como de procesamento, xa que todos eses datos precisan ser analizados para que a aplicación poida tomar decisións en base ao que aprendeu a partir deles. Incluso, podemos

ter que realizar dita análise en tempo real, por exemplo en tarefas de visión artificial como o recoñecemento facial. A operación clave no adestramento deste tipo de redes é a convolución. Se a aplicamos de forma iterativa a unha imaxe podemos ir extraendo as súas características, simplificando así o proceso de comparación, clasificación ou análise. En termos matemáticos, trátase dunha operación de álxebra lineal que involucra dúas matrices (ou tensores), que serán unha imaxe e un filtro, e produce unha nova matriz como resultado.

Por outro lado, para abordar a resolución deste tipo de operacións alxébricas, contamos coas vantaxes que nos ofrecen as plataformas heteroxéneas. Este tipo de sistemas están compostos por máis dun tipo de procesador ou núcleo, por exemplo unha CPU e unha GPU. As primeiras contan cunha unidade de control complexa e mecanismos de predicción de salto, co obxectivo de anticiparse aos acontecementos e aforrar ciclos na execución de instrucións. Ademais, están provistas de moi poucas unidades aritmético-lóxicas (ALU, *Arithmetic Logic Unit*) pero con moita potencia. Isto os converte en dispositivos extremadamente rápidos sempre e cando os datos deban ser procesados en orde secuencial. Pola contra, as tarxetas gráficas o que buscan é incrementar o rendemento xeral executando de forma simultánea unha mesma operación sobre múltiples datos. Para iso contan con moitos máis núcleos, aínda que máis simples e de baixo consumo, e evitan implementar mecanismos de control demasiado complexos. Aínda que inicialmente foron concebidas co propósito de renderizar imaxes, o certo é que a súa arquitectura SIMD (*Single Instruction Multiple Data*) vólveas especialmente axeitadas na resolución de problemas con gran paralelismo de datos. Este é o caso da convolución por lotes que, fronte a convolución tradicional, incrementa o número de datos a procesar aplicando un mesmo filtro a múltiples imaxes ao mesmo tempo. Esta nova forma de traballar coas GPUs denomínase GPGPU (*General-Purpose computing on Graphics Processing Units*) ou computación de propósito xeral en unidades de procesamento gráfico.

Finalmente, temos á nosa disposición distintos *frameworks*, linguaxes de programación e librarías para traballar con plataformas heteroxéneas. Nun traballo anterior [2] exploramos as capacidades que nos ofrecía CUDA, unha linguaxe propia das GPUs de NVIDIA. Nesta ocasión, non obstante, decantámonos por unha solución de Intel, oneAPI [3], que emprega a linguaxe de programación DPC++ (*Data Parallel C++*) [1].

1.2 Obxectivos

Como comentamos anteriormente, durante o adestramento das DNN, unha gran parte da carga de traballo recae sobre a operación de convolución. Existe un amplo panorama de investigación en relación coa paralelización do adestramento das DNN, e numerosos artigos que analizan ou presentan diversos algoritmos para implementar a operación de convolución. Algúns dos máis coñecidos son os baseados nos algoritmos de filtrado mínimo de Winograd,

na transformada de Fourier, ou na multiplicación de matrices. Ampliar o coñecemento sobre esta operación analizando a eficacia dos distintos métodos e implementándoos de forma paralela é unha das principais contribucións deste traballo. A continuación, concretaremos os seus obxectivos.

- Analizar o rendemento de implementacións xa existentes da convolución.

  O primeiro paso antes de comezar unha posible implementación consiste en realizar un traballo de investigación previo, coñecer as distintas opcións ata agora dispoñibles, comprendelas e probar o seu rendemento nunha plataforma heteroxénea. Para iso, faremos uso das distintas implementacións da convolución que ofrecen *frameworks* como a libraría oneDNN [4] incluída en Intel oneAPI, ou cuDNN de NVIDIA CUDA. A primeira inclúe os algoritmos GEMM e Winograd, mentres que a segunda, ademais, engade distintas variantes do FFT. Implementaremos unha serie de códigos de proba co obxectivo de realizar un estudo do rendemento dos algoritmos en distintas plataformas, CPUs e GPUs, e con problemas de distinto tamaño.

- Crear unha implementación en oneAPI da operación de convolución.

  Unha vez recoñecidos os algoritmos que ofrecen mellores resultados, procederemos a realizar unha implementación propia dalgunha outra variante que non estea implementada aínda en Intel oneAPI. Por exemplo, unha opción a valorar sería o algoritmo Coppersmith-Winograd para a multiplicación de matrices; outra, a do algoritmo Strassen no que está baseado o anterior. En todo caso, utilizaremos a linguaxe de programación DPC++ para o desenvolvemento do algoritmo, centrándonos durante a súa optimización nas arquitecturas paralelas. Concretamente, buscaremos explotar o maior rendemento posible empregando a GPU, e compararémolo cos resultados ofrecidos pola CPU.

- Estudar o rendemento das nova implementación e comparalo co rendemento doutras versións, secuenciais e paralelas.

  Finalmente, engadiremos os resultados que ofreza a nova implementación á análise de rendemento realizada con anterioridade. Unha parte moi importante do proceso consiste en estudar o rendemento do programa coas ferramentas provistas polo *framework* e desta forma dirixir a busca de posibles melloras, que serán implementadas de forma incremental.

1.3 Organización do proxecto

Nesta sección expoñemos os resultados do proxecto, os seus entregables; comprendendo como tales a ferramenta software (apartado 1.3.1) e a memoria do proxecto (apartado 1.3.2).

1.3.1 Ferramenta

Como resultado deste traballo, obteremos a implementación dun novo algoritmo e unha serie de conclusións, resultado dun proceso de estudo, aprendizaxe e investigación. Non se trata dun produto *software* final, senón unha peza de código estruturado nun repositorio que pode servir de base ou ser incluída en posteriores programas ou librarías. Este repositorio[1] segue unha estrutura lineal, en orde cronolóxica, onde os fitos máis relevantes foron marcados cunha etiqueta á que faremos alusión cando comentemos a evolución do proxecto.

1.3.2 Memoria

Esta memoria recolle o proceso de desenvolvemento levado a cabo durante o proxecto, o resultado das probas de rendemento pero tamén outras cuestións relativas á planificación e fundamentos teóricos do traballo. Esta introdución compón o primeiro de seis capítulos, que detallaremos a continuación.

Capítulo 2: Fundamentos teóricos e tecnolóxicos

No segundo capítulo plasmaremos os coñecementos técnicos adquiridos sobre as ferramentas e operacións involucradas no proxecto. Desta forma introduciremos a base teórica do traballo, necesaria para comprender os conceptos levados á práctica durante a implementación do mesmo.

Capítulo 3: Metodoloxía e plan de traballo

No terceiro capítulo comentaremos o método de traballo seguido e a planificación realizada. Ademais, realizaremos unha avaliación dos custos do proxecto.

Capítulo 4: Implementación

No cuarto capítulo detallaremos o proceso de implementación levado a cabo, explicando os algoritmos implementados, os problemas atopados e o procedemento seguido para solucionalos. Nas distintas seccións do capítulo explicaremos cada un dos bloques nos que se divide o código do proxecto. Dispoñerémolas en orde cronolóxica, co obxectivo de comprender a

[1] https://git.fic.udc.es/s.aguado/tfm

evolución dos distintos algoritmos implementados, para os que explicaremos o seu funcionamento e aquilo que os distingue dos anteriores.

Capítulo 5: Probas e resultados

Para rematar, no quinto capítulo analizaremos o rendemento obtido polos distintos algoritmos explicados no capítulo anterior. As probas aquí descritas serviron para obter unha métrica do rendemento que nos permitise iterar sobre os algoritmos e melloralos. Para dotar aos resultados de maior significado, debemos explicar tamén o contorno de execución das probas que, como veremos, se trata dunha plataforma de computación na nube.

Capítulo 6: Conclusións

Finalmente, no derradeiro capítulo, revisitaremos os fitos máis relevantes do proxecto, analizando a completude dos obxectivos, expoñendo o resultado final e valorando as posibles melloras.

Capítulo 2

Fundamentos teóricos e tecnolóxicos

Co obxectivo de poder comprender os conceptos levados á práctica durante a implementación da convolución por lotes, neste capítulo explicaremos os fundamentos teóricos desta operación e realizaremos unha introdución ás ferramentas utilizadas.

En primeiro lugar, na sección 2.1 "Fundamentos teóricos", explicaremos en que consiste a operación de convolución, cal é a súa aportación ás redes de aprendizaxe profunda e os distintos algoritmos que podemos empregar para implementala. En segundo lugar, na sección 2.2 "Fundamentos tecnolóxicos", faremos unha breve introdución a Intel oneAPI e a linguaxe DPC++, as ferramentas empregadas na paralelización dos algoritmos. Finalmente, na sección 2.3 "Ferramentas alternativas", expoñeremos aquelas tecnoloxías que tamén forman parte do ámbito do traballo ou que poderían substituír as aquí empregadas.

2.1 Fundamentos teóricos

Comezaremos a sección no apartado 2.1.1, explicando os fundamentos matemáticos da operación de convolución. Continuaremos no 2.1.2 comentando a súa aplicación nas redes de aprendizaxe profunda, e finalizaremos a sección no apartado 2.1.3 cunha relación de algoritmos que implementan a operación.

2.1.1 A convolución como operación matemática

A convolución é unha operación matemática que se define como a integral do produto entre dúas funcións continuas, f e g, tras desprazar unha delas unha distancia t. O resultado é unha terceira función $(f * g)$ que representa como a forma dunha modifica á outra.

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

Se nos movemos no eido do procesamento dixital de sinais, daquela a cada sinal corres-

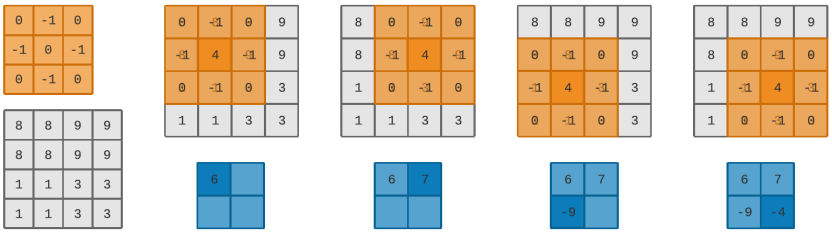

Figura 2.1: O resultado da convolución en dúas dimensións sobre unha imaxe (en gris) de 4x4 píxeles e un filtro (en laranxa) de 3x3 píxeles é outra imaxe (en azul) de 2x2 píxeles.

póndelle unha función discreta. Neste caso, a operación de convolución non varía, tan só é necesario substituír a integral por un sumatorio na ecuación:

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

Como podemos comprobar, esta primeira definición tan só contempla como entradas á función sinais dunha única dimensión. Agora ben, se o noso campo de traballo fora o procesamento e mellora de imaxes, poderíamos empregar a convolución para aplicar filtros sobre as mesmas. Cabe recordar que unha imaxe en branco e negro pode definirse como unha función de dúas dimensións, cuxa amplitude representaría a luminosidade dun píxel. Unha imaxe en cor, polo tanto, engadiría unha terceira dimensión que, neste caso, codificaría a información da cor.

Para comprender o proceso de filtrado dunha imaxe de dúas dimensións, imos empregar un exemplo gráfico. Na figura 2.1 podemos ver unha imaxe de 4x4 píxeles (en gris), e un filtro de 3x3 píxeles (en laranxa). O valor numérico de cada píxel corresponderíase co valor da luminosidade dese píxel dentro da escala de grises. Unha vez explicado isto, podemos comezar o filtrado da imaxe colocando o filtro sobre ela, na esquina superior esquerda. A continuación, multiplicamos os valores dos píxeles que se superpoñen e sumamos o resultado de cada multiplicación, o resultado deste sumatorio conforma o valor do primeiro píxel da imaxe de saída. O resto de pasos consisten en mover o filtro unha posición cara adiante e repetir o proceso, así ata completar todas as posicións da imaxe de saída.

Neste punto é posible que nos chame a atención que a imaxe de saída sexa máis pequena que a de entrada. En ocasións, co obxectivo de conservar as dimensións orixinais, realízase un preprocesado da imaxe de entrada que engade un marco (*padding*) de píxeles ao redor da imaxe. O tamaño deste marco depende das dimensións do filtro. É moi habitual que os píxeles engadidos teñan valor 0 (*zero-pad*), pero isto pode derivar nunha especie de halo escuro ao redor da imaxe de saída. Para evitalo, unha solución moi común consiste en replicar no marco

os valores do linde da imaxe.

Outras variantes do algoritmo introducen variables como o paso (*stride*) ou a dilatación (*dilation*). A primeira modifica o número de píxeles que salta o filtro en cada paso, é dicir, en lugar de avanzar unha única posición poderíamos saltar dous ou tres píxeles, segundo indique o *stride*. A segunda, no caso de ser distinta de cero, reduce o número de píxeles do filtro que participan na multiplicación. Neste traballo decidimos simplificar a operación e non utilizar estas variables, non obstante, no caso de querer afondar no asunto, aparecen moi ben explicadas no seguinte artigo [5].

2.1.2 Aplicacións nas redes de aprendizaxe profunda

Neste apartado veremos que relación teñen as redes de neuronas artificiais coa operación de convolución.

En primeiro lugar, unha rede neuronal artificial (ANN, *Artificial Neural Network*) pode ser definida como un modelo computacional formado por neuronas. As neuronas artificiais están baseadas -de forma conceptual- nas biolóxicas, pois se conectan formando unha rede con distintas capas e producen unha saída que serve como entrada á seguinte capa. Ademais, teñen unha serie de parámetros (pesos e *bias*) que poden ser alterados, outorgándolle á rede capacidade de aprendizaxe.

En segundo lugar, dentro desta definición entrarían as redes de aprendizaxe profunda (DNN, *Deep Neural Networks*), un tipo de rede neuronal que se diferencia por ter moitas máis capas do habitual. As principais vantaxes deste tipo de redes son: por un lado, que ademais de problemas lineares poden resolver relacións moito máis complexas e, por outro, que actúan sobre a totalidade dos datos sen necesidade de acción humana no proceso de extracción de características. Isto proporciónanos a capacidade de resolver problemas moi complexos, como por exemplo aqueles que implican a detección de obxectos en imaxes, sen necesidade de dispoñer de persoal especializado e, o máis importante, en tempo real.

En terceiro lugar, existen varios tipos de redes de aprendizaxe profunda. A continuación, faremos un breve repaso de cada variante, centrándonos naquela que máis interese ten para este traballo, i.e. as redes neuronais convolucionais.

RNN, *Recurrent Neural Networks*

Estas redes caracterízanse por ser retroalimentadas, é dicir, os datos de saída dunha capa son utilizados como entrada da mesma na seguinte iteración. Grazas a isto, teñen a capacidade de persistir certa información durante un tempo, coma se dun estado se tratase. Esta característica é de gran utilidade cando procesamos datos ligados por unha liña temporal, como pode ser o audio ou o vídeo. Polo tanto, son as redes máis indicadas en problemas como a síntese de voz, o procesamento de música ou da linguaxe.

UPN, *Unsupervised Pretrained Networks*

A diferencia do adestramento supervisado que adoita utilizar datos etiquetados por persoas (imaxinemos o típico problema de clasificación de animais), as redes non supervisadas son capaces de atopar patróns nun conxunto de datos sen etiquetas preexistentes. Entre elas podemos diferenciar distintos tipos, por exemplo:

- **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. É habitual a súa integración noutras redes de maior complexidade, como unha fase previa que se encarga de reducir a dimensionalidade dos datos.

- **DBN, *Deep Belief Networks***. Trátase dunha rede híbrida: por un lado, unha primeira fase utiliza máquinas de Boltzmann (RBM, *Restricted Boltzmann Machines*) [6] para realizar o adestramento sen supervisión; por outro, unha segunda fase formada pola típica rede neuronal [7] realiza o adestramento supervisado para refinar os resultados.

- **GAN, *Generative Adversarial Networks***. Caracterízanse por adestrar dous modelos ao mesmo tempo. Un deles, o modelo "xenerativo", sintetiza datos novos a partir dun gran conxunto de adestramento. O outro, o modelo "discriminante", consume eses datos 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.

CNN, *Convolutional Neural Networks*

As redes neuronais convolucionais deben o seu nome á operación de convolución. Mediante esta operación son capaces de extraer unha gran cantidade de información a partir dos datos de entrada, construíndo así un espazo de características moito máis robusto. Debido a isto, son especialmente axeitadas para resolver problemas de procesamento de imaxes, como o recoñecemento de obxectos en imaxes ou a visión por computador (tamén coñecida como visión artificial) que ten aplicacións directas no desenvolvemento de coches autónomos ou drones. De feito, adoitan ser as mellores nas aplicacións de clasificación e segmentación de imaxes.

Na figura 2.2 buscamos representar, de forma moi simplificada, a arquitectura dunha CNN, que consta dos seguintes bloques:

- Unha capa de entrada encargada de recibir os datos.

- Un gran número de capas convolucionais orientadas á extracción de características. Con este propósito poden participar varias funcións, as máis destacadas son:

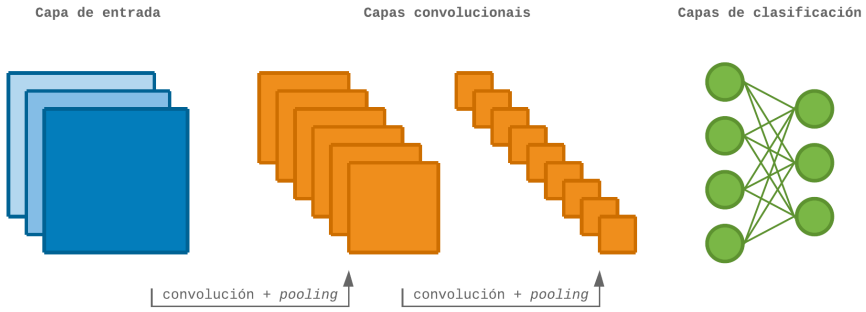

Figura 2.2: Bloques básicos da arquitectura dunha rede neuronal convolucional. Entre as funcións convolución e *pooling* utilízase unha función non linear, como a función ReLU.

  - A operación de convolución. Unha versión modificada da convolución 2D que explicaremos con maior detalle máis adiante.
  - Unha función de activación non lineal [8], como pode ser a función ReLU (*Rectified Linear Units*), cuxo propósito é acotar os valores de saída dunha capa antes de redirixilos como entradas á seguinte. Entre os beneficios que ofrece están: reducir o tempo de adestramento da rede e dotala de capacidade para resolver problemas non lineares.
  - Unha función de *pooling* [9] que reduce a dimensionalidade dos datos para simplificar o seu procesamento e evitar, entre outras cousas, o sobreadestramento.

- A arquitectura da rede finaliza cunha serie de capas de clasificación, unidas formando unha rede totalmente conectada (FC, *Fully-connected*). É habitual atopar redes que finalizan cunha función de activación, como a función *softmax*, co obxectivo de normalizar o resultado ofrecido polo clasificador.

A continuación, explicaremos o funcionamento da operación de convolución tal e como a interpretan as redes neuronais convolucionais.

En primeiro lugar, como entradas á operación, onde antes tiñamos imaxes en dúas dimensións (altura e largura, identificando as coordenadas dos píxeles) agora temos tensores de ata 4 dimensións. Como podemos consultar na táboa 2.1, a primeira dimensión (N) identifica o tamaño do lote (*batch*), a segunda (C) o número de canles da imaxe, mentres que as dúas últimas (H e W) continúan a representar a altura e a largura da imaxe, respectivamente.

En segundo lugar, a operación podería ser representada mediante a seguinte ecuación[1].

$$Y_{n,k,p,q} = \sum_{c}^{C} \sum_{r}^{R} \sum_{s}^{S} F_{k,c,r,s} \cdot X_{n,c,p+r,q+s} \tag{2.3}$$

[1] $\forall n \in [0, N) \quad \forall k \in [0, K) \quad \forall p \in [0, P) \quad \forall q \in [0, Q)$

N	Número de imaxes de entrada e de saída
K	Número de canles da imaxe de saída e, tamén, número de filtros
C	Número de canles da imaxe de entrada e do filtro
H	Número de filas da imaxe de entrada
W	Número de columnas da imaxe de entrada
R	Número de filas do filtro
S	Número de columnas do filtro
P	Número de filas da imaxe de saída
Q	Número de columnas da imaxe de saída

Táboa 2.1: Variables empregadas para denotar as dimensións dos tensores de entrada á función convolución.

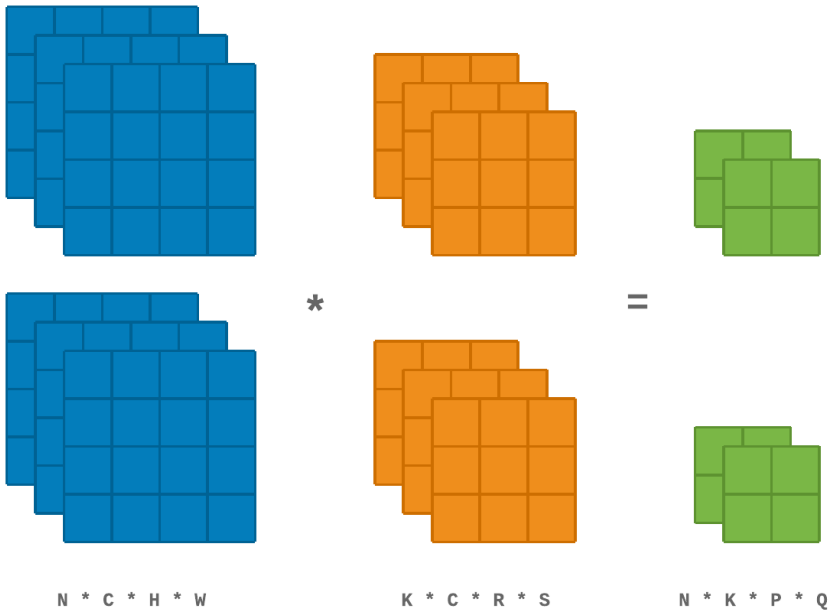

Figura 2.3: Unha CNN que tome como entradas dúas imaxes ($N = 2$) e dous filtros ($K = 2$) de tres canles cada un ($C = 3$), producirá dúas saídas (unha por imaxe) de dúas canles cada unha (unha por filtro).

Onde Y representa o tensor de saída, F os distintos filtros e X o tensor de entrada. O seu funcionamento consiste en aplicar a convolución 2D que explicamos no apartado 2.1.1 sobre a imaxe de entrada e todos filtros, obtendo como resultado un conxunto formado por K matrices bidimensionais de tamaño PxQ, onde cada unha representa unha canle da imaxe de saída. Se a imaxe de entrada e o filtro teñen máis dunha canle de entrada ($C > 0$) repetiremos a convolución en cada unha das canles e sumaremos as matrices resultantes entre si píxel a píxel. Ademais, se no lote hai máis dunha imaxe de entrada ($N > 0$) repetiremos os pasos anteriormente descritos con cada unha delas, obtendo en cada iteración unha nova imaxe de saída. Na figura 2.3 podemos ver de forma gráfica o cambio na dimensionalidade dos tensores tras pasar pola convolución. Para calcular as dimensións da imaxe de saída (as variables P e Q) podemos aplicar a seguinte fórmula sobre as dimensións da imaxe de entrada (H e W) e as do filtro (R e S):

$$P = (H - R + 2 \cdot padding_h)/stride_h + 1$$
$$Q = (W - S + 2 \cdot padding_w)/stride_w + 1$$

Onde *padding* e *stride* son as variables que xa explicamos no apartado 2.1.1 e que, por simplicidade, neste proxecto asumimos que sempre toman os valores 0 e 1, respectivamente.

Para rematar, agora que xa coñecemos a relevancia que ten a convolución no eido da intelixencia artificial, imos baixar o nivel e afondar nos distintos algoritmos dos que dispoñemos para implementala.

2.1.3 Algoritmos da convolución

Como sucede con calquera operación matemática, non hai un único xeito de implementar a convolución. As distintas variantes ofrecen distintas vantaxes, que poden estar relacionadas coa complexidade computacional, a compatibilidade con distintos formatos de entrada ou a simplicidade na súa implementación. A continuación, comentaremos os algoritmos máis comúns entre os *frameworks* que implementan a operación da convolución, como son o algoritmo directo e os baseados na multiplicación de matrices ou na transformada de Fourier.

Directo

Chamamos algoritmo directo a aquel que implementa a operación de convolución tal e como foi explicada no apartado 2.1.2. A forma máis sinxela de codificalo involucra sete bucles aniñados, que podemos ver representados no pseudocódigo da figura 1.

A pesar da sinxeleza da súa implementación, o gran problema deste algoritmo é a excesiva complexidade computacional que implica, e que volve inviable a súa aplicación en problemas do mundo real.

```
1  for n ← 0 to N do
2      for k ← 0 to K do
3          for c ← 0 to C do
4              for p ← 0 to P do
5                  for q ← 0 to Q do
6                      for r ← 0 to R do
7                          for s ← 0 to S do
8                              y[n][k][p][q] += f[k][c][r][s] * x[n][c][p+r][q+s];
9                          end
10                     end
11                 end
12             end
13         end
14     end
15 end
```

Algoritmo 1: Convolución directa

GEMM

Os algoritmos baseados na operación GEMM (*General Matrix Multiply*) [10] son uns dos máis empregados e gozan de especial relevancia neste proxecto. En xeral, constan de dúas operacións: por un lado, a primeira reordena os datos da imaxe de entrada de forma que, realizando unha multiplicación de matrices entre o filtro e a mesma, obteñamos o resultado da convolución; por outro lado, a multiplicación de matrices é unha operación que leva moito tempo sendo estudada, polo que existen numerosas optimizacións que podemos aproveitar para acelerar a convolución. Na figura 2.4 podemos ver unha representación gráfica do resultado da operación `im2col`, que reordena os datos de entrada seguindo as instrucións do pseudocódigo representado na figura 2.

Non obstante, esta aproximación ten un problema, pois o tamaño do espazo de traballo necesario para almacenar a imaxe transformada é moito maior que o tamaño da imaxe orixinal. En concreto, o espazo de traballo debe ter suficiente capacidade para almacenar N matrices de dimensións $(C \cdot R \cdot S)$ x $(P \cdot Q)$ cada unha. Por isto, algunhas librarías deciden elaborar formas alternativas de levar a cabo esta operación. É o caso da libraría BLIS (*BLAS-like Library Instantiation Software*) [11] que evita construír a matriz intermedia accedendo directamente aos valores da imaxe orixinal.

Respecto á multiplicación de matrices, dispoñemos de varios algoritmos que melloran a súa complexidade computacional, da orde de $O(n^3)$ na implementación máis básica. O primeiro en superar esta marca foi Volker Strassen [12]. Máis adiante, o algoritmo Coppersmith–Winograd[2] de 1990 [13] inspirou moitos outros algoritmos, incluíndo o máis rápido ata o momento descuberto por Josh Alman e Virginia Vassilevska no pasado ano 2020 [14].

[2]Non confundir cos algoritmos de filtrado mínimo ideados tamén por Shmuel Winograd

```
for n ← 0 to N do
    for c ← 0 to C do
        for p ← 0 to P do
            for q ← 0 to Q do
                for r ← 0 to R do
                    for s ← 0 to S do
                        workspace[n][c][r*S+s][p*Q+q] = x[n][c][p+r][q+s];
                    end
                end
            end
        end
    end
end
```

Algoritmo 2: Operación de transformación `im2col`

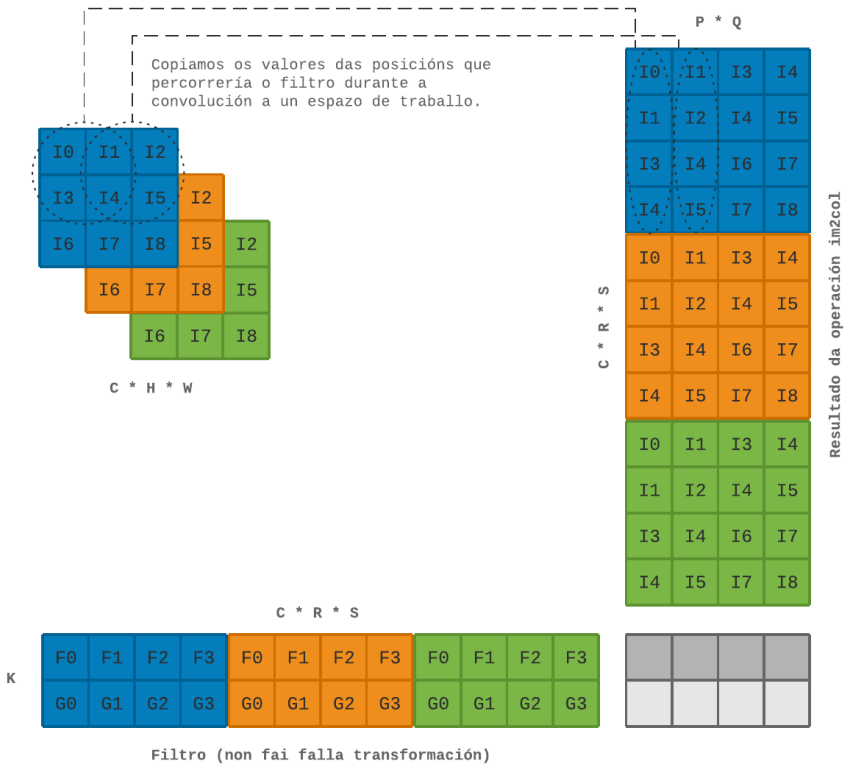

Figura 2.4: Mediante a operación *im2col* copiamos os valores das posicións da imaxe que percorrería o filtro durante a convolución a unha matriz que, multiplicada polo filtro, produce o resultado da propia convolución.

Winograd

Esta implementación está baseada nos algoritmos de filtrado mínimo que Shmuel Winograd plasmou no seu libro *Arithmetic complexity of computations* [15]. Os algoritmos orixinais estaban orientados a aplicacións no eido do procesamento dixital de sinais, como os filtros de resposta finita ao impulso (FIR, *Finite Impulse Response*). Non obstante, Andrew Lavin e Scott Gray realizaron unha reinterpretación destes algoritmos no artigo titulado *Fast Algorithms for Convolutional Neural Networks* [16]. Esta interpretación consiste en realizar certas transformacións sobre os datos de entrada para reducir ao mínimo o número total de multiplicacións a executar. A contrapartida é que o número de sumas necesarias aumenta de forma cuadrática con respecto ao tamaño da imaxe de entrada, por este motivo o algoritmo obtén os seus mellores resultados con moitas entradas pero de pequeno tamaño.

FFT

Os algoritmos baseados na transformada de Fourier rápida (FFT, *Fast Fourier Transform*) son unha técnica moi coñecida e empregada no procesamento de imaxes. Baséanse en trasladar o cálculo da convolución do dominio do tempo ao dominio da frecuencia, a través da transformada de Fourier, onde podemos expresalo coma un simple produto. Isto é posible grazas ao Teorema da Convolución [17], 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 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 procesar, xa que desta forma é posible reutilizar moitas das transformadas que se necesitan en cada convolución.

2.2 Fundamentos tecnolóxicos

Nesta sección realizaremos unha breve introdución ao *framework* Intel oneAPI e veremos as características clave da linguaxe de programación DPC++.

2.2.1 Introdución a Intel oneAPI

oneAPI [3] é un modelo de programación aberto, libre e estandarizado que busca mellorar o rendemento das aplicacións a través da súa paralelización, ao mesmo tempo que proporciona portabilidade entre os distintos tipos de dispositivos das plataformas heteroxéneas. Está composto por unha linguaxe de programación (DPC++) e unha serie de librarías (oneDNN, oneMKL, etc.) para crear aplicacións paralelas. O programador pode elixir entre utilizar DPC++ directamente para paralelizar un algoritmo, ou empregar unha implementación paralela xa dispoñible nalgunha das librarías incluídas.

A linguaxe de programación DPC++ (*Data Parallel C++*) [1] implementa o estándar SYCL [18] desenvolto polo grupo Khronos[3] sobre C++17, tamén estandarizado pola norma ISO [19]. A maiores, engade algunhas extensións propias. Unha das características clave desta linguaxe é que permite empregar unha única implementación entre dispositivos heteroxéneos (CPUs, GPUs e FPGAs) a traveś dunha interface común.

Na figura 2.5 podemos ver a estrutura básica dun programa SYCL/DPC++. Para poder entender este exemplo e os que explicaremos ao longo do proxecto, nos restantes apartados faremos un repaso dos conceptos básicos desta linguaxe. Comezaremos por explicar como se realiza a selección do dispositivo no que queremos que sexa executado o código (apartado 2.2.2), a continuación veremos uns conceptos básicos sobre memoria e xestión das estruturas de datos (apartado 2.2.3), e finalmente explicaremos que son os *kernels*, que tipos hai e como definilos (apartado 2.2.4).

2.2.2 Selección do dispositivo

Nun ficheiro fonte DPC++ podemos atopar dous tipos de código: o código do anfitrión (*host*) e o código do dispositivo (*device*). O primeiro execútase na CPU principal da máquina, e o segundo nun dispositivo secundario como, por exemplo, unha GPU. A comunicación entre ambos dispositivos lévase a cabo a través de colas (*queues*).

Queue

Unha cola é unha abstracción á que se poden subscribir accións para seren executadas nun único dispositivo. Ditas accións adoitan ser o lanzamento dun *kernel*, é dicir, unha peza de código paralelo executada no dispositivo. Non obstante, tamén nos permiten realizar tarefas máis complexas, por exemplo, en lugar de deixar que o *runtime* realice o movemento dos datos de forma automática, podemos facelo manualmente a través de funcións que dispoñen as colas. Os dous métodos máis importantes da clase `queue` son `submit` e `wait`:

- `submit`: envía á cola unha función *lambda* cos comandos a executar no dispositivo.

[3] https://www.khronos.org/

```
#include <CL/sycl.hpp>
#include <iostream>
#define N 8

using namespace sycl;

int main() {

  int *array = new int[N];
  for (int i = 0; i < N; i++) array[i] = i;

  {
    // Create a QUEUE to the default_selector
    queue q;

    // Create a one-dimensional BUFFER using host allocated array
    buffer b(array, range(N));

    q.submit([&](handler &h) {

      // Get an ACCESSOR to the host data
      accessor a(b, h, write_only);

      // Execute the KERNEL in the device
      h.parallel_for(N, [=](auto &i) {
        a[i] += 1;
      });
    });
  }

  // The array will be updated upon exiting the scope
  for (int i = 0; i < N; i++)
    std::cout << "array[" << i << "] = " << array[i] << "\n";

  return 0;
}
```

Figura 2.5: Estrutura básica dun programa SYCL.

- `wait`: realiza unha espera bloqueante ata a finalización das tarefas encoladas.

É importante recordar que unha cola só pode estar ligada a un único dispositivo, por este motivo o método máis común para elixir o dispositivo no que serán executadas as accións é a través do constructor da cola. Na figura 2.5 declaramos unha cola sen indicar ningún parámetro no constructor, desta forma estamos dicíndolle ao *runtime* que pode escoller calquera dispositivo dispoñible. SYCL/DPC++ garante que sempre vai haber polo menos un dispositivo dispoñible, i.e. o propio anfitrión, que tamén pode executar *kernels*. Non obstante, se precisamos un dispositivo ou tipo en concreto, podemos engadir ao constructor da cola un selector de dispositivo (*device selector*).

Device Selector

Aquelas clases que implementan a clase abstracta `device_selector` son coñecidas como selectores de dispositivo. O estándar inclúe cinco selectores:

- `default_selector`: selecciona calquera dispositivo.
- `host_selector`: selecciona o dispositivo anfitrión.
- `cpu_selector`: selecciona un dispositivo que se identifique como CPU.
- `gpu_selector`: selecciona un dispositivo que se identifique como GPU.
- `accelerator_selector`: selecciona calquera dispositivo exceptuando CPUs e GPUs.

A maiores, DPC++ engade dous selectores máis:

- `INTEL::fpga_selector`: selecciona un dispositivo que se identifique como FPGA.
- `INTEL::fpga_emulator_selector`: selecciona unha CPU que emule unha FPGA.

Se, por exemplo, quixésemos executar o *kernel* da figura 2.5 nunha GPU, necesitaríamos enviar o selector de dispositivo como parámetro no constructor da cola, tal e como amosa a seguinte liña de código:

```
queue q( gpu_selector{} );
```

Outro método interesante para coñecer os detalles do dispositivo seleccionado é `get_info`:

```
std::cout << q.get_device().get_info<sycl::info::device::name>();
```

2.2.3 Xestión de datos

No apartado anterior vimos como controlar a execución de código en distintos dispositivos. Como cabe a posibilidade de que ditos dispositivos non compartan memoria, precisamos un xeito de mover os datos dun a outro. Con este propósito, SYCL/DPC++ proporciona tres formas distintas de comunicar as distintas memorias: a *Unified Shared Memory* (USM), os *buffers* e as imaxes (*images*). A continuación veremos con maior detalle cada un destes tipos.

Unified Shared Memory

A USM (*Unified Shared Memory*) é unha abstracción empregada para acceder a memoria distribuída a través de punteiros, tal e como faríamos en C/C++. Debido a esta característica, é especialmente útil cando precisamos migrar grandes cantidades de código dende esta linguaxe. Para lograr este efecto, a USM emprega un único espazo de enderezos virtual, polo que un punteiro froito dunha reserva de memoria no anfitrión será válido para a súa utilización en calquera dispositivo. Toda reserva de memoria neste espazo de enderezos virtual debe ser iniciada no anfitrión mais, segundo o seu modo, pode ter distintas características:

- `device`: Memoria reservada no dispositivo e non accesible para o anfitrión. Os datos deben ser copiados de forma explícita.
- `host`: Memoria reservada no anfitrión e accesible para todos os dispositivos de forma implícita.
- `shared`: Aplican tamén as propiedades comentadas no anterior punto. A diferencia clave entre a memoria `host` e a memoria `shared` é que, no momento de ser accedida dende un dispositivo, a primeira sempre vai derivar en accesos remotos[4], mentres que a segunda copiará os datos no primeiro acceso e o resto realizaranse de forma local.

Como adiantábamos hai un intre, temos dúas formas de copiar os datos da memoria do anfitrión á memoria do dispositivo. A máis sinxela é empregar os modos `host` ou `shared` e deixar que o *runtime* decida o momento de copiar os datos (figura 2.6). Non obstante, se precisamos facer cousas máis complexas, como superpoñer a copia dos datos ao procesamento dos mesmos, podemos empregar o tipo `device` e realizar a copia de forma manual a través do método `memcpy` da clase `handler` (figura 2.7).

[4]Mentres que un **acceso local** ten como destino a memoria conectada directamente ao dispositivo. Un **acceso remoto** é aquel que involucra datos aloxados na memoria doutro dispositivo. Os primeiros son máis lentos porque os datos deben atravesar enlaces de maior latencia e/ou menor ancho de banda.

```
#include <CL/sycl.hpp>
#include <iostream>
#define N 8

using namespace sycl;

int main() {

  // Create a QUEUE passing a device_selector
  queue q(gpu_selector{});

  int *shared_array = malloc_shared<int>(N,q);
  int *host_array = malloc_host<int>(N,q);
  for (int i = 0; i < N; i++) host_array[i] = i;

  // Execute the KERNEL in the device
  q.submit([&](handler &h) {
    h.parallel_for(N, [=](auto &i) {
      shared_array[i] = host_array[i] + 1;
    });
  }).wait();

  for (int i = 0; i < N; i++)
    std::cout << "array[" << i << "] = " << shared_array[i] << "\n";

  return 0;
}
```

Figura 2.6: Exemplo de uso da *Unified Shared Memory* cos modos `host` e `shared` e realizando unha copia implícita dos datos.

```
#include <CL/sycl.hpp>
#include <iostream>
#define N 8

using namespace sycl;

int main() {

  // Create a QUEUE passing a device_selector
  queue q(gpu_selector{});

  int *device_array = malloc_device<int>(N,q);
  int *host_array = new int[N];
  for (int i = 0; i < N; i++) host_array[i] = i;

  // Copy the initialized data from host to device
  q.submit([&](handler &h) {
    h.memcpy(device_array, host_array, N*sizeof(int));
  }).wait();

  // Execute the KERNEL in the device
  q.submit([&](handler &h) {
    h.parallel_for(N, [=](auto &i) {
      device_array[i] += 1;
    });
  }).wait();

  // Copy the result data from device to host
  q.submit([&](handler &h) {
    h.memcpy(host_array, device_array, N*sizeof(int));
  }).wait();

  for (int i = 0; i < N; i++)
    std::cout << "array[" << i << "] = " << host_array[i] << "\n";

  return 0;
}
```

Figura 2.7: Exemplo de uso da *Unified Shared Memory* co modo `device` e realizando unha copia explícita dos datos.

Buffers

A outra abstracción da que dispoñemos para acceder a memoria distribuída son os *buffers*. Un `buffer` é un envoltorio para un ou máis obxectos dalgún tipo C/C++. A forma máis sinxela de crealos é utilizando un punteiro á memoria do anfitrión e indicando o número de elementos que contén nun obxecto `range` de ata 3 dimensións:

```
buffer b(array_3d, range(X,Y,Z));
```

Ao contrario da USM, os *buffers* non representan posicións de memoria, polo que non poden ser accedidos coma se fosen punteiros. No seu lugar, para ler e escribir dun *buffer* é necesario declarar un obxecto *accessor*. Para maior optimización, podemos indicar o tipo de uso que lle vamos a dar ao *buffer* a través de modificadores de acceso:

```
accessor a(b, h, write_only);
```

Desta forma, utilizaremos a etiqueta `write_only` cando só vaiamos escribir, `read_only` cando só vaiamos ler do *buffer* e `read_write` cando pensemos en realizar ambas operacións. Unha vez definido o *accessor* poderemos empregalo coma se fose un punteiro aos elementos contidos no *buffer*.

```
a[0] = a[1] + a[2];
```

Para rematar, se regresamos á figura 2.5, poderemos ver un exemplo completo do uso de *buffers* e *accessors*. A maiores, se nos fixamos veremos que o código SYCL/DPC++ está enclaustrado entre dúas chaves, comezando na liña 12 e rematando na 29. Estas chaves están definindo un ámbito (*scope*) igual que o fan outros recursos da linguaxe coma o `try-catch`. Nesta ocasión realizan unha función de sincronización entre dispositivo e anfitrión; de forma que, ao saír do ámbito, o *runtime* actualiza as estruturas de datos modificadas no dispositivo para que o anfitrión poida acceder aos datos modificados. Como experimento, podemos probar a eliminar ditas chaves, e veremos que o programa imprime o *array* tal e como foi inicializado no *host*, sen os cambios introducidos polo dispositivo.

Images

As imaxes (*images*) son un tipo especial de *buffer* que proporciona funcionalidades específicas para o procesamento de imaxes. O seu funcionamento é moi similar ao que acabamos de explicar, e o seu propósito moi específico, polo que non afondaremos máis neste tema.

2.2.4 Definición de kernels

Un *kernel* é unha peza de código que pode ser executada nun dispositivo. Grazas a eles os programadores podemos esquecer os detalles de implementación das distintas plataformas e utilizar unha interface común a todas. En SYCL/DPC++ temos distintas formas de definir un *kernel*: funcións *lambda*, obxectos de función (*functors*), binarios ou código en formato *string* ao estilo de OpenCL. Por ser consistentes co código implementado neste proxecto, nos exemplos desta introdución empregaremos sempre funcións *lambda*.

As expresións *lambda* foron introducidas en C++11 e serven para crear funcións anónimas. A sintaxe dunha función *lambda* é a seguinte:

```
[ captures ] ( params ) -> return-type { body }
```

As expresións contidas entre corchetes (*captures*) definen o modo de acceso ás variables que, aínda que non formen parte dos parámetros (*params*) da función, seguen podendo ser utilizadas no corpo da mesma. Para o noso caso de estudo chega con saber que se a *lambda* comeza por `[&]` as variables capturadas serán pasadas por referencia, mentres que se comeza por `[=]` serán pasadas por valor. No caso dos *kernels*, a captura deberá ser sempre por valor.

Como vén sendo habitual, existen tres tipos de *kernels*: o primeiro tipo é perfecto para unha primeira aproximación paralela, mais non permite controlar os detalles de baixo nivel; o segundo engade opcións para personalizar e optimizar o noso *kernel*, e o terceiro tipo proporciona unha sintaxe alternativa que pretende simplificar a definición de *kernels* multidimensionais a través dunha estrutura xerárquica.

Kernels básicos

Este tipo de *kernels* ofrecen unha interface de programación moi sinxela, mais non deixan marxe á personalización dos detalles de baixo nivel. A asignación dos recursos hardware ás distintas instancias do *kernel* está integramente en mans do *runtime* e, segundo medra a complexidade do *kernel*, máis complicado se volve entender os seus resultados en canto a rendemento se refire.

Utilizalos é tan sinxelo como chamar á función `parallel_for` da clase `handler`. Esta función recibe dous argumentos: un obxecto `range` onde indicamos o número de elementos de procesamento (instancias do *kernel*) que queremos lanzar, e unha función *lambda* que deberá executar cada elemento de procesamento. No seguinte exemplo podemos ver un *kernel* que realiza a suma de vectores dunha única dimensión.

```
h.parallel_for(range{N}, [=](id<1> idx) {
  c[idx] = a[idx] + b[idx];
});
```

Tamén podemos aproveitar os rangos multidimensionais para escribir código que involucre estruturas de máis dunha dimensión. Por exemplo, para paralelizar unha multiplicación de matrices poderíamos usar o seguinte *kernel*.

```
h.parallel_for(range{M,N}, [=](id<2> idx) {

  int m = idx[0];
  int n = idx[1];

  for (int k = 0; k < K; k++) {
    c[m][n] += a[m][k] * b[k][n];
  }
}
```

Como é posible observar nos exemplos, neste punto toma especial relevancia a clase `range`, que xa utilizamos á hora de declarar os *buffers*. A continuación, realizaremos unha explicación máis detallada desta clase, así como dos obxectos `id` e `item`.

- A clase **`range`** representa un rango de ata tres dimensións. O número de dimensións do rango debe ser coñecido en tempo de compilación, mais o número de elementos en cada dimensión podemos pasarllo ao constructor en tempo de execución. Como xa vimos no apartado 2.2.3, ademais de definir a dimensionalidade do *kernel*, a clase `range` tamén pode indicar as dimensións dos *buffers*.

- A clase **`id`** representa un índice dentro dun rango de ata tres dimensións. Podemos imaxinalo como un *array* de enteiros con, como máximo, tres posicións. Accedendo ás distintas posicións podemos recuperar o noso *id* dentro de cada dimensión.

- A clase **`item`** encapsula os obxectos `range` e `id` para aqueles *kernels* aos que non lles chega con coñecer a súa posición no rango, senón que tamén precisan saber o tamaño das dimensións. Con este fin expón funcións para obter o `id` (`get_id`) e o rango (`get_range`) así coma outras propiedades da instancia do *kernel* que se está a executar.

No interior do *kernel* podemos ter acceso aos obxectos de tipo `id` ou `item` sempre e cando os declaremos como parámetros da función *lambda*.

Kernels ND-range

A característica máis importante deste tipo de *kernels* é que permiten crear grupos (e subgrupos) de varias instancias. Isto nos ofrece certo control sobre a localidade dos datos, así como sobre a asignación de recursos hardware ás distintas instancias do *kernel*. Na figura 2.8 podemos ver unha representación gráfica dos distintos grupos nos que podemos dividir

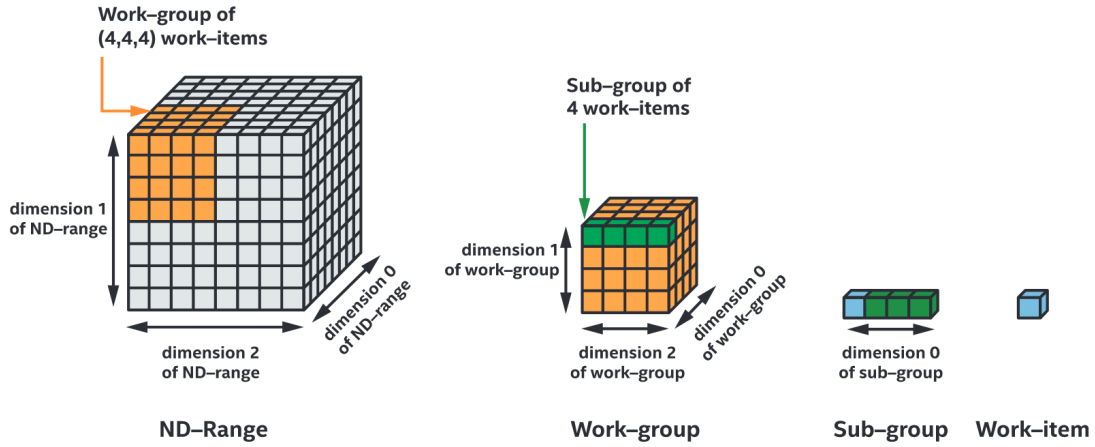

Figura 2.8: Rango de execución dun *kernel ND-range* de tres dimensións dividido en grupos de procesamento (*work-groups*), subgrupos (*sub-groups*) e, finalmente, elementos de procesamento (*work-items*). Imaxe: *Reinders et al. 107* [1]

as múltiples instancias do *kernel*. A continuación definiremos os tres novos elementos que entran en xogo.

- *work-item*: cada elemento de procesamento representa unha instancia do código do *kernel*. Nun *kernel* básico, sen grupos de traballo, non hai forma de sincronizar nin comunicar os *work-items* entre si.
- *work-groups*: os *work-items* dun mesmo grupo de traballo poden comunicarse a través de funcións colectivas e dispoñen dunha memoria local, que nalgúns dispositivos se corresponde cunha memoria dedicada máis rápida que a global. Existe un número máximo de *work-items* por *work-group* dependendo do dispositivo, podemos obtelo a través do método `get_info` da clase `device`.
- *sub-groups*: os *work-items* dun mesmo subgrupo poden comunicarse a través de funcións colectivas, igual que facían dentro dun *work-group*. A principal diferencia é que os dispositivos con características SIMD poden asignarlles un réxime de execución especial, por exemplo, empregando funcións vectoriais. Os subgrupos, a diferencia dos grupos de traballo, só teñen unha única dimensión e o seu tamaño depende do dispositivo, do *kernel* e incluso do *ND-range* que definamos. Podemos obter este valor a través do método `get_info` da clase `kernel`.

O seguinte código mostra como poderíamos reescribir a multiplicación de matrices que víamos antes empregando un *kernel ND-range*.

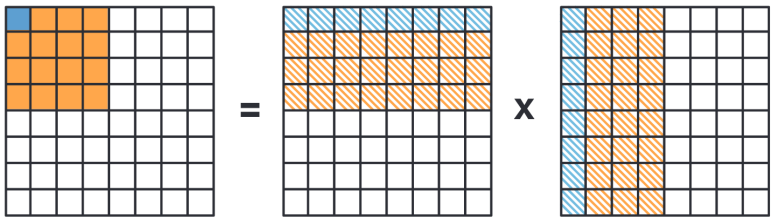

Figura 2.9: Asignación de *work-items* (azul) e *work-groups* (laranxa) aos elementos a procesar nunha multiplicación de matrices. Imaxe: *Reinders et al. 113* [1]

```
range global{M,N};
range local{B,B};

h.parallel_for(nd_range{global,local}, [=](nd_item<2> it) {

  int m = it.get_global_id(0);
  int n = it.get_global_id(1);

  for (int k = 0; k < K; k++) {
    c[m][n] += a[m][k] * b[k][n];
  }
});
```

Como podemos observar, a principal diferencia entre as dúas é que agora temos dous rangos: un global que indica o número de *work-items* en cada dimensión do *nd-range*, e un local que indica o número de *work-items* en cada dimensión do *work-group*. Polo demais, só temos que substituir as clases `range` e `item` polas súas análogas `nd-range` e `nd-item`. A vantaxe desta implementación reside en que desta forma aproveitamos moito mellor a localidade dos datos.

Para comprendelo mellor, imos fixarnos no exemplo da figura 2.9, onde escollemos un rango local de 4x4 elementos de procesamento. Nun *kernel* básico, cada *work-item* tería que acceder a memoria global para ler os datos que lle corresponden. Quere dicir, que se temos 16 elementos de procesamento, realizaríamos 16 accesos a memoria global. Non obstante, neste *kernel ND-range*, de 16 elementos de procesamento que hai en cada *work-group*, só 4 deles teñen que acceder a memoria, o resto poden ler os datos directamente da caché.

Kernels xerárquicos

Os *kernels* xerárquicos (*hierarchical data-parallel kernels*) ofrecen unha sintaxe alternativa para os *kernels ND-range*, onde cada nivel de paralelismo se representa aniñando unha nova chamada á función `parallel_for`. O obxectivo desta sintaxe era asemellala á programación de bucles paralelos, non obstante introduce certas particularidades que dificultan o traballo

do compilador e, de feito, aínda está en fase experimental. Debido a isto, tan só imos mostrar un exemplo para coñecer a sintaxe, mais non imos afondar nos detalles.

```
range num_groups{N/B,N/B};
range group_size{B,B};

h.parallel_for_work_group(num_groups, group_size, [=](group<2> grp) {
  int mb = grp.get_id(0);
  int nb = grp.get_id(1);

  grp.parallel_for_work_item([&](h_item<2> it) {
    int m = mb*B + it.get_local_id(0);
    int n = nb*B + it.get_local_id(1);

    for (int k = 0; k < K; k++) {
      c[m][n] += a[m][k] * b[k][n];
    }
  });
});
```

2.3 Ferramentas alternativas

A continuación, faremos mención a distintas ferramentas que entran dentro do ámbito do proxecto mais non forman parte do mesmo. Tales coma outros *frameworks* para a programación de plataformas heteroxéneas (apartado 2.3.1), librarías que optimicen rutinas recorrentes no *Deep Learning* (apartado 2.3.2) ou que aceleran operacións de álxebra lineal (apartado 2.3.3).

2.3.1 Linguaxes ou *frameworks* para plataformas heteroxéneas

Neste traballo decidimos empregar o *framework* Intel oneAPI, o cal xa introducimos na sección anterior. Non obstante, existen outras ferramentas que tamén nos permiten programar plataformas heteroxéneas. A continuación propoñemos algunhas delas, realizando un breve resumo das mesmas.

NVIDIA CUDA

NVIDIA Compute Unified Device Architecture [20] é unha plataforma de computación paralela que permite realizar computación de propósito xeral (GPGPU, *general-purpose computing on graphics processing units*) en GPUs de NVIDIA. Trátase dun *framework* deseñado para interactuar dende C/C++ ou Fortran e, a diferencia doutras linguaxes de programación en GPUs

como OpenGL[5], non require de coñecementos avanzados en programación de dispositivos gráficos. O seu modelo de programación baséase na definición de *kernels* pequenas pezas de código que son executadas de forma concorrente en numerosos núcleos da GPU, a través das cales temos acceso ao conxunto de instrucións da gráfica.

OpenCL

Open Computing Language [21] é un estándar aberto e mantido polo grupo Khronos. Conforma unha interface de programación (API, *Application Programming Interfaces*) orientada a dispositivos tan diversos como CPUs, GPUs, procesadores de sinais dixitais (DSP, *Digital Signal Processors*) ou FPGAs (*Field-Programmable Gate Arrays*), entre outros. Ao tratarse dunha interface, é responsabilidade dos fabricantes realizar a correspondente implementación do estándar, habendo implementacións da gran maioría dos fabricantes, véxase Intel, NVIDIA, Apple (hoxe en día obsoleto xunto con OpenGL), ARM, IBM, Qualcomm, e un longo etc.

2.3.2 Librarías que optimizan bloques básicos do *Deep Learning*

Aínda que un dos nosos obxectivos neste traballo era realizar unha implementación propia do algoritmo da convolución, o certo é que xa existen librarías que nos permiten realizar este tipo de operacións. Por exemplo:

Intel oneDNN

Anteriormente coñecida como MKL-DNN ou *Deep Neural Network Library* (DNNL), Intel oneDNN [4] é unha libraría de código aberto que busca mellorar o rendemento das aplicacións de *Deep Learning* optimizando os bloques principais que conforman as redes de aprendizaxe profunda. Está optimizada para procesadores e tarxetas gráficas de Intel, o soporte de arquitecturas ARM ou NVIDIA de momento é experimental. Cabe destacar que aplicacións de *Deep Learning* tan coñecidas como PyTorch[6] ou Tensorflow[7] empregan esta libraría para acelerar as súas rutinas. Entre todas as funcións que inclúe, no ámbito deste traballo destacan os tres algoritmos da convolución que implementa:

- Winograd - Unha versión do algoritmo explicado no apartado 2.1.3, baseada no traballo *Fast Algorithms for Convolutional Neural Networks* (Lavin et al.) [16].
- Direct - Unha optimización do algoritmo directo centrada no uso de instrucións SIMD.
- GEMM - Algoritmo baseado na operación GEMM, tamén explicado no apartado 2.1.3.

[5] https://www.opengl.org/
[6] https://pytorch.org/
[7] https://www.tensorflow.org

NVIDIA cuDNN

A libraría *NVIDIA CUDA Deep Neural Network* [22] optimiza, mediante a paralelización con CUDA, algoritmos moi utilizados no adestramento e inferencia de redes de aprendizaxe profunda. Está pensada para ocultar os detalles de máis baixo nivel, optimizando os bloques máis empregados, pero deixando o resto da implementación en mans do programador. Entre as funcións que implementa atópanse os algoritmos máis coñecidos e utilizados da convolución: GEMM, FFT e Winograd, incluíndo algunha variante dos mesmos.

2.3.3 Librarías que optimizan operacións de álxebra lineal

Na sección previa nomeamos aquelas librarías que implementan bloques funcionais das redes de aprendizaxe profunda, por exemplo, a función convolución. Non obstante, se quixésemos implementar un algoritmo non contemplado nalgunha destas librarías, mais delegando os detalles da súa optimización nunha libraría especializada en operacións alxébricas, poderíamos escoller entre as seguintes.

Intel MKL

A libraría *Intel Math Kernel Library* [23], procura ofrecer unha interface de programación sinxela e, ao mesmo tempo, incrementar o rendemento das aplicacións mediante a optimización de rutinas matemáticas básicas. Soporta todo tipo de arquitecturas x86 de 32 e 64 bits e inclúe optimizacións para todas as familias de procesadores Intel, incluíndo os procesadores *manycore* Intel Xeon Phi. Está dispoñible de forma gratuíta para sistemas Linux, Windows e macOS en tres linguaxes de programación distintas: C/C++, Fortran e Python a través de adaptadores.

NVIDIA cuBLAS

Mentres que MKL está optimizada para CPUs de Intel, a libraría *NVIDIA CUDA Basic Linear Algebra Subprograms* [24] tan só é compatible con GPUs de NVIDIA. Está pensada para ser integrada de forma sinxela en programas CUDA e aporta unha implementación das operacións de álxebra lineal máis comúns optimizadas para este tipo de plataformas.

BLIS

A libraría BLIS (*BLAS-like Library Instantiation Software*) [25] parte da refactorización dun proxecto actualmente inactivo, GotoBLAS (no que tamén se basean outras librarías coma OpenBLAS), e busca reducir a cantidade de código necesario para programar unha plataforma concreta. Só é compatible coas CPUs máis modernas.

Capítulo 3

Metodoloxía e plan de traballo

Neste terceiro capítulo, estableceremos o método de traballo a seguir durante o desenvolvemento do proxecto e realizaremos unha planificación coas tarefas a completar. Debido ao carácter evolutivo do entorno de traballo que envolve aos proxectos de investigación, é moi importante escoller unha metodoloxía de desenvolvemento axeitada. Desta forma, evitaremos imprevistos e conseguiremos realizar unha xestión eficaz do esforzo e do tempo de traballo.

3.1 Metodoloxía

O traballo que temos entre mans caracterízase por ser un proxecto de investigación, onde o produto final non está completamente definido e polo tanto é presumible de sufrir cambios. É por iso que decidimos deseñar o plan de traballo seguindo unha metodoloxía de desenvolvemento áxil, que nos permita ir planificando as tarefas en pequenas iteracións, co obxectivo de construír o resultado final de forma incremental e evitando, ademais, sobrecargas na planificación. A evolución do traballo será supervisada de forma periódica, ao final de cada iteración, a través de reunións entre o titor e a estudante. Nestas reunións farase unha retrospectiva das tarefas completadas e planificarase o traballo futuro.

Outro punto a ter en conta é o tamaño do equipo que, nesta ocasión, vese reducido a un único desenvolvedor e un director. O director, Diego Andrade, conta cunha ampla experiencia no campo do HPC e as plataformas heteroxéneas, e ten dirixido e participado en proxectos que empregan as mesmas tecnoloxías. Sobre el recaerá a coordinación do equipo de traballo a través das xa citadas reunións. Pola súa parte, a estudante realizará o traballo de investigación e desenvolvemento do código, así como o de redacción da memoria. Conta con experiencia previa na optimización de rutinas do *Deep Learning* e a programación en GPUs (CUDA). Ámbolos dous traballaron de forma conxunta nun proxecto anterior, e contan coa formación necesaria para cumprir os obxectivos desta investigación.

3.2 Planificación

Aínda que execución do traballo tome forma iterativa podemos definir, en base aos obxectivos do proxecto, unha serie de tarefas clave que deben ser completadas. A continuación, mostramos a relación destas tarefas en orde cronolóxica, acompañadas dunha breve descrición:

T1 Estudar o *framework* Intel oneAPI e aprender os conceptos básicos da linguaxe de programación DPC++. Realizar unha implementación do algoritmo directo da convolución nesta linguaxe. O obxectivo é dispoñer dunha liña base coa que comparar a mellora que supoñen o resto de algoritmos fronte á implementación máis simple desta operación.

Entregables. Código DPC++ coa implementación do algoritmo da convolución. Ademais da redacción dun informe sobre o método de traballo con Intel oneAPI.

T2 Medir o rendemento de distintas implementacións da convolución. Probar os diferentes algoritmos presentes en librarías como oneDNN.

Entregables. Memoria dos experimentos realizados, onde se explique a natureza das probas e se expoñan os resultados acadados, i.e. o rendemento e a eficiencia dos distintos algoritmos tanto na CPU como na GPU.

T3 Explorar outras alternativas á realización da operación que nos atangue. Nesta tarefa o director terá un papel moi relevante, orientando a dirección que tomará o proxecto.

Entregables. A estudante redactará un informe no que se perfilen as principais características dos distintos algoritmos valorados e se xustifique a escolla do algoritmo polo que finalmente nos decantamos.

T4 Implementar o algoritmo escollido en Intel oneAPI. Comezar cunha versión funcional pero ineficiente e ir iterando sobre ela ata acadar un resultado aceptable. Utilizar as ferramentas que proporciona o *framework* para atopar posibles melloras.

Entregables. Código coa implementación do algoritmo, explicación detallada do seu funcionamento e informe co proceso de desenvolvemento seguido.

T5 Estudar o rendemento da nova implementación e comparalo cos resultados acadados polos demais algoritmos. Neste punto, recorrerase ao servizos *cloud* de Intel, "DevCloud for oneAPI", para ter acceso a infraestruturas HPC e mellorar a calidade dos resultados.

Entregables. Informe cos resultados dos experimentos, de forma que se amplíe a información recollida no informe da tarefa nº2.

T6 Documentación do traballo realizado.

Entregables. Memoria final co resumo do proxecto e as conclusións acadadas, ademais do enderezo ao repositorio co código fonte.

Para realizar a planificación, asumiremos que o tempo de traballo será de 4h ao día. Isto quere dicir que, sendo a duración do proxecto un total de 300h, este se estenderá durante 75 días comprendidos entre febreiro e xuño do ano 2021. Na figura 3.1 representamos unha estimación do tempo que levarían as distintas tarefas e o seu reparto sobre o calendario. As zonas en color gris representarían os días nos que sería imposible traballar no proxecto debido a factores externos a el, como actividades relacionadas co mestrado e prácticas en empresa. Mentres que o venres 25 de xuño aparece marcado en negro porque se trata da data límite para a presentación do proxecto.

Finalmente, comprendemos que a interdependencia entre as tarefas entraña un risco para a finalización en tempo do proxecto. Xa que, se algunha delas se atrasa, a data de finalización do proxecto tamén se pode ver afectada. Non obstante, para evitar este tipo de imprevistos, dividiremos as tarefas principais en ciclos máis pequenos que nos permitan solucionar de forma rápida os posibles problemas atopados. Ademais, non descartamos a posibilidade de aumentar o tempo de traballo diario como medida de continxencia.

3.3 Avaliación de custos

A continuación, realizaremos unha estimación dos custos económicos do traballo. En primeiro lugar, cómpre partirmos dunha relación na que se detallen os recursos do proxecto, é dicir: persoal, materiais e software necesario para o completo desenvolvemento do traballo. Ditos recursos son:

- Materiais:
  - Acceso a un PC ou clúster de gama alta onde poder realizar probas de rendemento.
  - Un PC de gama media/baixa con CPU e/ou GPU de marca Intel onde levar a cabo o desenvolvemento.
- Software:
  - *Intel oneAPI Base Toolkit* [1], ou máis concretamente dúas ferramentas incluídas neste: a libraría *Intel oneAPI Deep Neural Network Library (oneDNN)* e o compilador *Intel oneAPI DPC++/C++ Compiler (dpcpp)*.

[1] https://software.intel.com/content/www/us/en/develop/tools/oneapi/base-toolkit

1	2	3	4	5	6	7	Febreiro
8	9	10	11	12	13	14	
15	16	17	18	19	20	21	
22	23	24	25	26	27	28	
1	2	3	4	5	6	7	Marzo
8	9	10	11	12	13	14	
15	16	17	18	19	20	21	
22	23	24	25	26	27	28	
29	30	31	1	2	3	4	Abril
5	6	7	8	9	10	11	
12	13	14	15	16	17	18	
19	20	21	22	23	24	25	
26	27	28	29	30	1	2	Maio
3	4	5	6	7	8	9	
10	11	12	13	14	15	16	
17	18	19	20	21	22	23	
24	25	26	27	28	29	30	
31	1	2	3	4	5	6	Xuño
7	8	9	10	11	12	13	
14	15	16	17	18	19	20	
21	22	23	24	25	26	27	
28	29	30					
T1	T2	T3	T4	T5	T6	Total	Tarefa
13	10	2	25	10	15	75	Días
17%	13%	3%	33%	13%	20%	100%	Porcentaxe

Figura 3.1: Estimación da duración de cada tarefa nunha representación sobre o calendario

Recurso	Prezo unitario	Tempo	Total
Programadora	20 €/h	300 h	6000 €
Xefe de proxecto	30 €/h	75 h	2250 €
Equipo gama media/baixa	16.6 €/mes	5 meses	83 €
Intel DevCloud	-	3 meses	0 €
Software	-	-	0 €
Total			8333 €

Táboa 3.1: Estimación do custo de cada recurso e suma total do proxecto

- Persoal:
  - Alumna: formará parte do equipo de investigación e de traballo, realizando as tarefas de deseño, implementación e avaliación de resultados.
  - Titor: como investigador principal, será o encargado de dirixir o proxecto e coordinar o equipo de traballo.

En segundo lugar, avaliaremos o custo unitario de cada recurso e a cantidade dos mesmos para poder calcular o custo total do proxecto, que ascendería a 8333€, a maior parte man de obra. Dito cálculo pode ser consultado na táboa 3.1.

En último lugar, cómpre realizar unha serie de aclaracións para comprender a natureza dos datos presentes na táboa. Por un lado, en relación ao número de horas de traballo da alumna e o titor, tomáronse como referencia os valores presentes na guía docente[2]. Por outro lado, a plataforma escollida para a realización das probas de rendemento foi *Intel DevCloud for oneAPI*[3], un entorno especialmente escollido para o desenvolvemento de aplicacións HPC con ferramentas Intel. Ofrécese gratuitamente por un período de tres meses, ampliable ata un ano a petición do desenvolvedor. Finalmente, para calcular a amortización realizada sobre o equipo informático, tomouse como referencia un prezo base de 800€ a repartir entre catro anos de vida útil. Polo tanto, podemos calcular o custo mensual do equipo do seguinte modo:

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

Independentemente do número de horas traballadas, a duración total do proxecto foi de aproximadamente cinco meses dende a data de inicio ata a data de finalización do mesmo.

[2] https://guiadocente.udc.es/guia_docent/index.php?assignatura=614473111&fitxa_apartat=4
[3] https://software.intel.com/content/www/us/en/develop/tools/devcloud.html

Capítulo 4

Implementación

A o longo deste capítulo describiremos o proceso de desenvolvemento levado a cabo durante o proxecto. Como adiantábamos en capítulos anteriores, o produto software obtido como resultado deste traballo está composto por unha serie de códigos que implementan distintos algoritmos. Todos os aquí expostos poden atoparse no repositorio de código fonte[1] do proxecto, onde tamén dispoñemos de instrucións para compilar e executar as probas. Para organizar os ficheiros fonte, utilizamos unha estrutura de directorios, de forma que as distintas versións dun mesmo algoritmo dispoñan do seu propio cartafol. Desta forma, podemos dividir o proxecto en tantos bloques como algoritmos. Nas distintas seccións do capítulo trataremos cada un destes bloques, en orde cronolóxica, indicando os pasos que foron necesarios para o deseño e implementación dos algoritmos.

En primeiro lugar, na sección 4.1 aproveitaremos a convolución directa para explicar cuestións transversais ao resto de algoritmos. En segundo lugar, na sección 4.2 comentaremos as diferentes probas implementadas sobre a libraría oneDNN. Esta será a única sección que non trate sobre unha implementación propia, senón que avalíe os algoritmos xa incluídos na libraría en cuestión. En terceiro lugar, na sección 4.3 explicaremos a implementación do algoritmo baseado na operación GEMM; e finalmente, en cuarto e último lugar, a sección 4.4 dedicarémola á variante do algoritmo GEMM empregada pola libraría BLIS.

4.1 Algoritmo directo

Na planificación realizada no capítulo 3, a primeira tarefa consistía en realizar unha introdución ao *framework* Intel oneAPI, co obxectivo de familiarizarnos coa linguaxe de programación DPC++. Para iso, nun primeiro momento comezamos realizando un programa que implementa a convolución en dúas dimensións, seguindo o algoritmo explicado no apartado 2.1.1 do capítulo 2. Para verificar que o resultado da operación é correcto, comparámolo co

[1] https://git.fic.udc.es/s.aguado/tfm

resultado dunha execución secuencial do mesmo. Podemos localizar esta versión, marcada coa etiqueta `2d-conv`[2], no repositorio. Unha vez realizada esta primeira toma de contacto, o seguinte paso consiste en mudar ao algoritmo que realmente nos interesa: o algoritmo da convolución directa explicado no apartado 2.1.3. Esta implementación, que explicaremos con maior detalle a continuación, podemos atopala no directorio `src/direct`[3] do repositorio.

4.1.1 Xeneralidades

De forma xeral, da implementación de cada algoritmo obteremos dous ficheiros fonte. Poñendo como exemplo este primeiro caso, teremos: `direct_sequential.cpp` coa implementación secuencial do algoritmo e `direct_parallel.cpp` coa implementación en DPC++. Ademais, entra en xogo un terceiro ficheiro común a todas as implementacións: `utils.hpp`, que busca simplificar a lectura dos ficheiros principais reunindo as tarefas comúns a todos os programas; tales como a inicialización das estruturas de datos, o procesamento dos parámetros de entrada ou a verificación dos resultados.

A interface de todos os programas tamén é común: por un lado poderemos elixir o dispositivo no que executar o programa, CPU ou GPU; por outro lado, tamén poderemos indicar as dimensións dos tensores de entrada, é dicir, os parámetros N, C, K, H, W, R e S, cuxo significado podemos consultar na táboa 2.1. Para velo dun modo algo máis gráfico, poderíamos dicir que a interface dos distintos executables segue o seguinte patrón:

```
./executable (cpu|gpu) N C K H W R S
```

Onde os parámetros da convolución poden tomar valores enteiros e positivos. Cabe destacar que as versións secuenciais, aínda que acepten o primeiro parámetro (dispositivo) por compatibilidade, só poden ser executadas na CPU. Finalmente, para simplificar o proceso de compilación, os ficheiros fonte de cada algoritmo veñen cun ficheiro Makefile preparado para realizar dous tipos de compilación:

- *all.* Por defecto o programa realiza unha execución silenciosa, é dicir, sen escribir ningún resultado na saída estándar. Este é o modo que utilizamos á hora de executar as probas de rendemento. Se algún erro fose detectado durante o programa, este si que se mostraría, para evitar confusións na toma de tempos.

- *debug.* Este é o modo que usamos cando queremos comprobar que o resultado numérico do programa é correcto e mostrar certa información por pantalla. Os cambios no comportamento do programa en función do modo de compilación son xestionados a través da *macro* DEBUG, introducida como parámetro na secuencia de compilación.

[2] https://git.fic.udc.es/s.aguado/tfm/tags/2d-conv
[3] https://git.fic.udc.es/s.aguado/tfm/tree/master/src/direct

Xa para finalizar, podemos compilar todos os exemplos ao mesmo tempo executando un *script* de nome `build` situado no directorio raíz do repositorio, que creará o cartafol `bin` onde meterá todos os executables.

4.1.2 En detalle

Unha vez explicadas as cuestións transversais, imos afondar na estrutura do programa que implementa de forma paralela o algoritmo directo da convolución: `direct_parallel.cpp`.

Por un lado, o *main* é unha función dunha única instrución que inicializa as dimensións da convolución en función dos parámetros de entrada e, a continuación, executa a función *convolution*. Este é o modelo que seguirán tamén o resto de programas.

```
1 int main(int argc, char **argv) {
2   return handle_errors(parse_arguments(argc,argv), convolution);
3 }
```

Por outro lado, a función *convolution* comeza declarando e inicializando os tensores, e remata coa comprobación dos resultados. Como único parámetro toma unha variable que serve para seleccionar o dispositivo no que vai ser executado o programa. Non imos afondar nos detalles da codificación, mais podemos afirmar que o programa segue a estrutura dun programa SYCL que empregue *buffers* como método de acceso a memoria, tal e como vimos na figura 2.5 do capítulo 2. O máis interesante desta implementación é a función co código do dispositivo, o *kernel*. Na figura 4.1 podemos ver unha versión simplificada do *kernel* deste algoritmo. Como podemos observar, con esta función podemos chegar a calcular cada unha das posicións do tensor de saída de forma concorrente. Outro detalle importante é que, como sabemos, tanto os *buffers* coma os obxectos de tipo *range* teñen como moito tres dimensións. Isto implica que para acceder a estruturas de maior dimensionalidade, coma é o caso dos tensores, debemos empregar aritmética de punteiros. Co índice sucede o mesmo, nas liñas 5 e 6 da figura 4.1 estamos calculando as posicións p e q a partir do valor do índice (de cero a $P \cdot Q$) e a dimensión principal Q.

4.2 Probas con oneDNN

A seguinte tarefa da nosa planificación tiña como obxectivo recoller métricas do rendemento dos algoritmos implementados na libraría oneDNN. Con este fin estudamos a documentación da libraría en cuestión para coñecer, en primeiro lugar, os algoritmos da convolución que implementa e, en segundo lugar, como integrala no noso programa para poder utilizar ditos algoritmos. Comezaremos esta sección explicando os conceptos básicos da programación con oneDNN, poñendo como exemplo a aplicación realizada para a execución das probas, e despois comentaremos as primitivas da convolución que implementa e como foron

```
context.parallel_for(sycl::range(N,K,P*Q), [=](auto index) {

  int n = index[0];
  int k = index[1];
  int p = index[2] / Q;
  int q = index[2] % Q;

  for (int c = 0; c < C; c++) {
    for (int r = 0; r < R; r++) {
      for (int s = 0; s < S; s++) {
        y[n][k][p][q] += x[n][c][p+r][q+s] * f[k][c][r][s];
      }
    }
  }
});
```

Figura 4.1: Simplificación do *kernel* paralelo do algoritmo directo da convolución

aplicadas no programa. Podemos atopar o código deste bloque no directorio `src/onednn`[4] do repositorio.

4.2.1 Modelo de programación

oneDNN é unha libraría baseada en contexto, como tamén o son cuDNN e outras librarías orientadas á programación de plataformas heteroxéneas. O seu modelo de programación consiste na execución de primitivas (*primitives*) sobre un ou máis obxectos de memoria (*memory objects*). Esta execución pode realizarse en distintos dispositivos (*engine*) e baixo un determinado contexto (*stream*). Na figura 4.2 podemos ver un esquema que relaciona as distintas entidades que acabamos de mencionar. Con esta estrutura en mente podemos comezar a analizar o noso programa.

En primeiro lugar, debemos crear os obxectos `dnnl::engine` e `dnnl::stream` que definen o noso contexto de execución. O parámetro `engine_kind` é unha variable de tipo `dnnl::engine::kind` que tomará o valor `cpu` ou `gpu` en función do que o usuario indicara no primeiro argumento de entrada ao programa.

```
dnnl::engine engine(engine_kind, 0);
dnnl::stream stream(engine);
```

En segundo lugar, temos que declarar os obxectos de memoria das estruturas de datos que van participar da convolución. Para iso necesitamos o *engine* no que van ser utilizados e un descritor de memoria. Os descritores son obxectos que almacenan os *metadatos* dos obxectos de memoria, é dicir, as súas dimensións (`dnnl::memory::dims`), o tipo de dato que van almacenar (`dnnl::memory::data_type`) e o formato no que se van gardar os datos

[4] https://git.fic.udc.es/s.aguado/tfm/tree/master/src/onednn

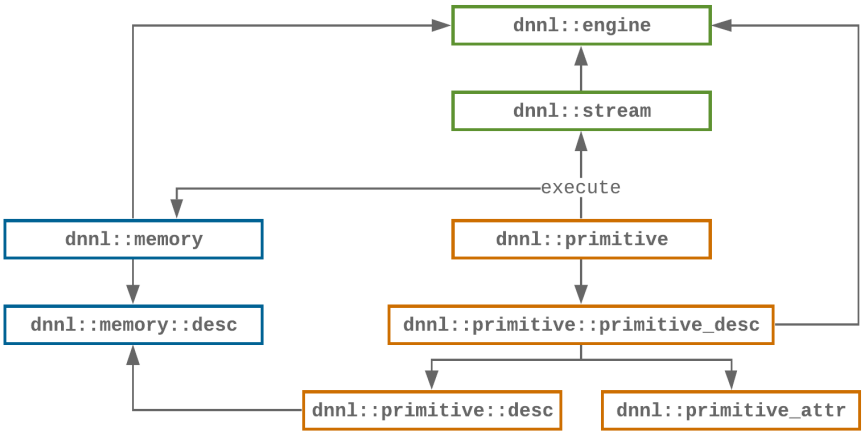

Figura 4.2: Dependencias entre os distintos obxectos da libraría oneDNN. O obxectivo é executar no dispositivo (obxectos en cor verde) unha primitiva (laranxa) que pode tomar certas estruturas de memoria (azul) como argumentos.

(`dnnl::memory::format_tag`). Unha vez declarados, poderemos proceder coa copia dos datos do anfitrión ao dispositivo.

```
dnnl::memory::desc x_desc({N,C,H,W}, type::f32, format::nchw);
dnnl::memory x_mem(x_desc, engine);
```

En terceiro lugar, só quedaría declarar a primitiva da convolución, unha variable de tipo `dnnl::primitive`. Unha vez máis, o construtor da primitiva ínstanos a definir un descritor (`dnnl::primitive::primitive_desc`). O construtor desta clase admite tres parámetros: o primeiro é outro descritor, `dnnl::primitive::desc`; o segundo un obxecto de tipo `dnnl::primitive_attr` co que podemos indicar operacións adicionais, a executar antes ou despois da primitiva, por exemplo a operación ReLU; finalmente, como terceiro parámetro indicamos o *engine* que declaramos anteriormente.

```
dnnl::convolution_forward::desc conv_desc(
  dnnl::prop_kind::forward_inference,
  dnnl::algorithm::convolution_direct,
   x_desc, f_desc, b_desc, y_desc, // memory descriptors
   {SH,SW}, {PH_L,PW_L}, {PH_R,PW_R} // stride and padding dimensions
);
dnnl::convolution_forward conv_prim({conv_desc, {}, engine});
```

No noso programa realizamos certo traballo adicional para convertir os datos do formato que nós utilizamos ao formato que utiliza a primitiva, e viceversa. Desta forma asegurámonos de estar realizando unha comparación xusta entre os distintos algoritmos.

Para rematar, invocamos ao método `execute` da primitiva, que toma como parámetros o *stream* e un dicionario cos argumentos da convolución, ou o que vén sendo o mesmo, cos

obxectos de memoria dos tensores.

```
conv_prim.execute(stream, {
  {DNNL_ARG_SRC,     x_mem},
  {DNNL_ARG_WEIGHTS, f_mem},
  {DNNL_ARG_BIAS,    b_mem},
  {DNNL_ARG_DST,     y_mem}
});
```

O único que quedaría por facer sería copiar o resultado da convolución da memoria do dispositivo á memoria do anfitrión.

4.2.2 Algoritmos da convolución

Neste programa, cuxo funcionamento acabamos de resumir, somos capaces de mudar o algoritmo a utilizar mediante a definición de distintas *macros* en tempo de compilación. Para aprender a invocar os distintos algoritmos implementados en oneDNN, acudimos á súa documentación[5]. Segundo nela indican, a día de hoxe oneDNN implementa tres algoritmos diferentes: o directo, Winograd e o baseado na operación GEMM; dos cales só os dous primeiros poden ser seleccionados de forma explícita.

- `convolution_winograd` - Algoritmo Winograd baseado no traballo *Fast Algorithms for Convolutional Neural Networks* (Lavin et al.) [16]. Aínda que o programador o seleccione de forma explícita, se non se cumpren unhas condicións deriva a execución ao seguinte algoritmo, `convolution_direct`.

- `convolution_direct` - Usa instrucións SIMD para optimizar o algoritmo directo. Igual que o anterior, tamén teñen que cumprirse certas condicións para que sexa executado, se non se cumpren, executa o `convolution_gemm`.

- `convolution_gemm` - Transforma a imaxe de entrada nunha matriz bidimensional e realiza unha multiplicación de matrices. Non se pode seleccionar manualmente, trátase do algoritmo que se executa automaticamente se non se cumpren as condicións dos demais.

Non obstante, no noso programa fomos capaces de forzar a execución do algoritmo GEMM a vontade, mediante o incumprimento das condicións deliberadamente, e tamén de lanzar unha excepción cando se intente usar Winograd nun dispositivo non compatible.

[5] https://docs.oneapi.com/versions/latest/onednn/dev_guide_convolution.html#algorithms

4.3 Algoritmo GEMM

Como resultado da terceira tarefa da planificación, tomamos a decisión de implementar unha versión optimizada do algoritmo GEMM. Non obstante, antes de comezar o estudo desta optimización (que veremos na sección 4.4) implementaremos primeiro o algoritmo básico. Seguindo o exemplo do primeiro bloque, o da convolución directa, neste caso tamén realizaremos dúas versións do algoritmo: comezando cunha versión secuencial que despois paralelizaremos en DPC++. O código deste bloque podemos atopalo no directorio `src/gemm`[6] do repositorio.

4.3.1 Secuencial

Como xa explicamos no apartado 2.1.3 do capítulo 2, o algoritmo GEMM consta de dúas operacións: por un lado, a operación `im2col`, que realiza a reordenación da imaxe de entrada; e por outro, a multiplicación de matrices, que podemos identificar no código C++ (`gemm_sequential.cpp`) como unha función de nome `matmul`. Esta versión secuencial non entraña ningún misterio, simplemente repite as dúas operacións mencionadas con cada unha das imaxes do lote. Na que temos que poñer maior atención é na versión DPC++, que veremos a continuación.

4.3.2 Paralelo

O ficheiro `gemm_parallel.cpp` contén unha implementación en DPC++ do algoritmo da convolución GEMM. Para realizar a paralelización deste algoritmo son necesarios dous *kernels*, un por cada operación. Se nos fixamos nas figuras 4.3 e 4.4 podemos obter unha visión simplificada de ambos *kernels*. É moi importante que o resultado da primeira operación estea listo para cando comece a segunda, por iso realizamos dúas chamadas ao método `submit` da clase `queue`. Tras a primeira chamada, empregamos o método `wait` para sincronizar dispositivo e anfitrión. Unha vez remata a execución no dispositivo, o anfitrión abandona a espera e envía o segundo *kernel* á cola.

O maior inconveniente deste algoritmo é a gran cantidade de memoria que utiliza para almacenar a matriz intermedia, que volve inviable a súa utilización en problemas relativamente grandes. Para evitar este problema, algunhas librarías, como é o caso da libraría BLIS, realizan unha implementación do algoritmo que evita construír a matriz problemática accedendo directamente ás posicións de memoria da imaxe orixinal coa que se corresponde. Na seguinte sección veremos a nosa aproximación a esta versión mellorada do algoritmo GEMM, que chamaremos GEMM-BLIS.

[6] https://git.fic.udc.es/s.aguado/tfm/tree/master/src/gemm

```
device_queue.submit([&](handler &context) {

  accessor x(x_buf, context, read_only);
  accessor b(b_buf, context, write_only);

  context.parallel_for(range(N,C,R*S), [=](auto index) {

    int n = index[0];
    int c = index[1];
    int r = index[2] / S;
    int s = index[2] % S;

    for (int p = 0; p < P; p++) {
      for (int q = 0; q < Q; q++) {
        b[n][c][r*S+s][p*Q+q] = x[n][c][p+r][q+s];
      }
    }
  });
}).wait();
```

Figura 4.3: Vista simplificada do *kernel* da operación `im2col`

```
device_queue.submit([&](handler &context) {

  accessor f(f_buf, context, read_only);
  accessor b(b_buf, context, read_only);
  accessor y(y_buf, context, write_only);

  context.parallel_for(range(N,K,P*Q), [=](auto index) {

    int n = index[0];
    int i = index[1];
    int j = index[2];

    for (int k = 0; k < C*R*S; k++) {
      y[n][i][j] += f[i][k] * b[n][k][j];
    }
  });
});
```

Figura 4.4: Vista simplificada do *kernel* da multiplicación de matrices

4.4 Algoritmo GEMM-BLIS

A cuarta tarefa da nosa planificación consistía en implementar unha mellora ao algoritmo da convolución en Intel oneAPI. Como adiantábamos no apartado anterior, a mellora escollida foi unha versión do algoritmo GEMM que reduce o consumo de memoria evitando construir a matriz intermedia. Como esta tamén é a aproximación utilizada na libraría BLIS, decidimos nomeala GEMM-BLIS. Como vén sendo habitual, implementamos dúas versións deste algoritmo: unha secuencial e outra paralela, cuxo funcionamento explicaremos a continuación. Podemos atopar o código deste bloque no directorio `src/blis`[7] do repositorio.

4.4.1 Secuencial

Para elaborar o programa `blis_sequential.cpp` inspirámonos no algoritmo descrito no traballo *High Performance and Portable Convolution Operators for Multicore Processors* (San Juan et al.) [11]. O funcionamento deste algoritmo consiste en utilizar dous *buffers* temporais, A e B, para almacenar en cada momento un cacho de filtro e outro de matriz, respectivamente. A continuación, multiplica as matrices contidas en ditos *buffers* para obter unha parte do resultado. Desta forma, pouco a pouco vai completando a imaxe de saída.

Na figura 3 representamos o pseudocódigo deste algoritmo, no que podemos ver como se empaquetarían as entradas nos *buffers*, e que posicións do resultado axudarían a construír. Como é posible observar, o código está dividido en dous bloques. O primeiro está formado por tres bucles, que recorren as estruturas de datos empaquetando as matrices de entrada nos *buffers* $Ac[MC][KC]$ e $Bc[KC][NC]$, cuxa multiplicación dará lugar a unha parte do resultado, $Cc[MC][NC]$. O segundo bloque está formado por dous bucles que volven utilizar dous novos *buffers*, $Ar[MR][KC]$ e $Br[KC][NR]$, máis pequenos ca os anteriores. Cabe destacar que estes últimos non son estritamente necesarios, o obxectivo de usar unha estrutura xerárquica é aproveitar os distintos niveis de memoria caché da máquina obxectivo. Para iso deberíamos escoller o tamaño dos *buffers* de forma que poidamos almacenar as matrices Ac e Bc na caché de segundo nivel (L2), e as matrices Ar e Br na caché de primeiro nivel (L1). A modo de aclaración, na figura 4.5 representamos un exemplo de como poderíamos empaquetar os tensores de entrada en *buffers* de menor tamaño.

Xa para finalizar, de entre todas as funcións auxiliares a este algoritmo, a máis importante é a que crea a matriz Bc a partir da imaxe de entrada, ao mesmo tempo que aplica a operación `im2col` sobre unha pequena parte desta. Podemos ver o funcionamento desta operación no código da figura 4.6.

[7] https://git.fic.udc.es/s.aguado/tfm/tree/master/src/blis

```
1  for jc ← 0 to N step NC do
2      for pc ← 0 to K step KC do
3          for ic ← 0 to M step MC do
4              Ac ← A[ic:ic+MC-1][pc:pc+KC-1]
5              Bc ← B[pc:pc+KC-1][jc:jc+NC-1]
6              Cc ← C[ic:ic+MC-1][jc:jc+NC-1]
7              for jr ← 0 to NC step NR do
8                  for ir ← 0 to MC step MR do
9                      Cc[ir:ir+MR-1][jr:jr+NR-1]
10                         += Ac[ir:ir+MR-1][0:KC-1]
11                            * Bc[0:KC-1][jr:jr+NR-1]
12                 end
13             end
14         end
15     end
16 end
```

Algoritmo 3: Pseudocódigo do algoritmo GEMM-BLIS, versión secuencial

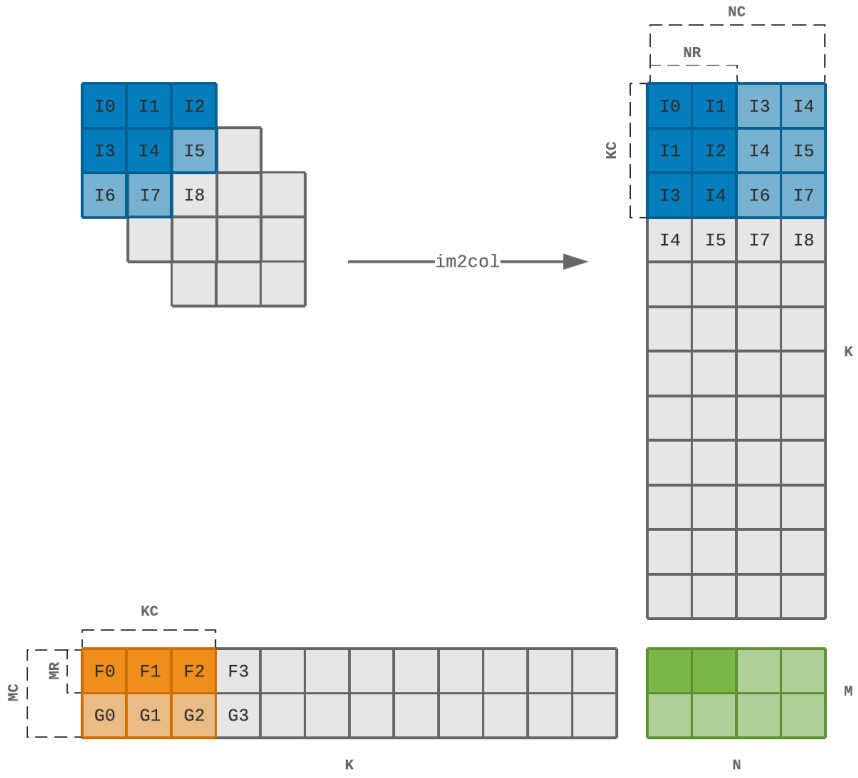

Figura 4.5: Exemplo coa distribución de distintos *buffers* sobre as matrices que participan no algoritmo GEMM-BLIS

```
for (int ps = 0; ps < KC; ps++) {
  int c =  (pc+ps)/RS;
  int r = ((pc+ps)%RS)/R;
  int s = ((pc+ps)%RS)%R;

  for (int js = 0; js < NC; js++) {
    int p = ((jc+js)%PQ)/P;
    int q = ((jc+js)%PQ)%P;

    Bc[ps][js] = B[c][p+r][q+s];
  }
}
```

Figura 4.6: Función de empaquetado da matriz Bc, que substitúe a operación `im2col`

4.4.2 Paralelo

Para elaborar a versión paralela, `blis_parallel.cpp`, cambiamos lixeiramente o paradigma. En lugar de utilizar dous *buffers* bidimensionais para almacenar unha parte da matriz intermedia, utilizamos un único *buffer* por elemento de procesamento, no que cada proceso almacena unha columna da matriz. Isto facémolo de tal xeito que aproveitemos o que en SYCL/DPC++ se coñece como *memoria privada* dos elementos de procesamento. Un tipo de memoria máis rápida que a principal, como pode ser unha caché, e que depende do dispositivo.

Na figura 4.7 podemos ver unha simplificación do *kernel* desta versión que, xunto co diagrama da figura 4.8, demostra o funcionamento do algoritmo. No código podemos ver que o *buffer* non ten por que ter a lonxitude exacta dunha columna, de feito, para poder almacenalo na memoria privada ten que ter un tamaño constante. Por este motivo, o corpo do *kernel* é un bucle que vai recorrendo a columna en segmentos da lonxitude do *buffer*; aínda que, na práctica, o tamaño do *buffer* pode ser o suficientemente grande como para almacenar a columna completa. As principais vantaxes desta estratexia son dúas. Por un lado, desta forma paralelizamos o procesamento de cada columna do tensor de saída, de forma que cada fío só ten que procesar K elementos (tantos como o número de filtros). Por outro lado, utilizando o *buffer* para almacenar unha única columna aproveitamos moito máis a localidade dos datos, pois estamos copiando en posicións de memoria contiguas aqueles datos que van ser accedidos (ata en K ocasións) de forma secuencial.

En conclusión, grazas a esta optimización podemos aplicar o algoritmo GEMM sobre matrices de maior tamaño, pois o espazo de traballo necesario xa non aumenta de forma proporcional ao tamaño do problema. Non é de estrañar que esta sexa a aproximación que mellores resultados ofrece, xa que con ela realizamos unha mellor xestión dos recursos *hardware* do dispositivo e aproveitamos a localidade dos datos. Poderemos comprobalo no próximo capítulo, onde discutiremos o rendemento das distintas implementacións descritas neste.

```
context.parallel_for(sycl::range(P,Q,N), [=](auto index) {
  float Bc[SIZE];

  int p = index[0];
  int q = index[1];
  int n = index[2];

  int jc = p*Q*N + q*N + n;
  for (int pc = 0; pc < C*R*S; pc += SIZE) {
    int kc = MIN(SIZE, C*R*S-pc);

    Bc <- B[pc:pc+kc-1][jc];
    C[0:K-1][jc] += A[0:K-1][pc:pc+kc-1] * Bc;
  }
});
```

Figura 4.7: Simplificación do *kernel* do algoritmo GEMM-BLIS, versión paralela

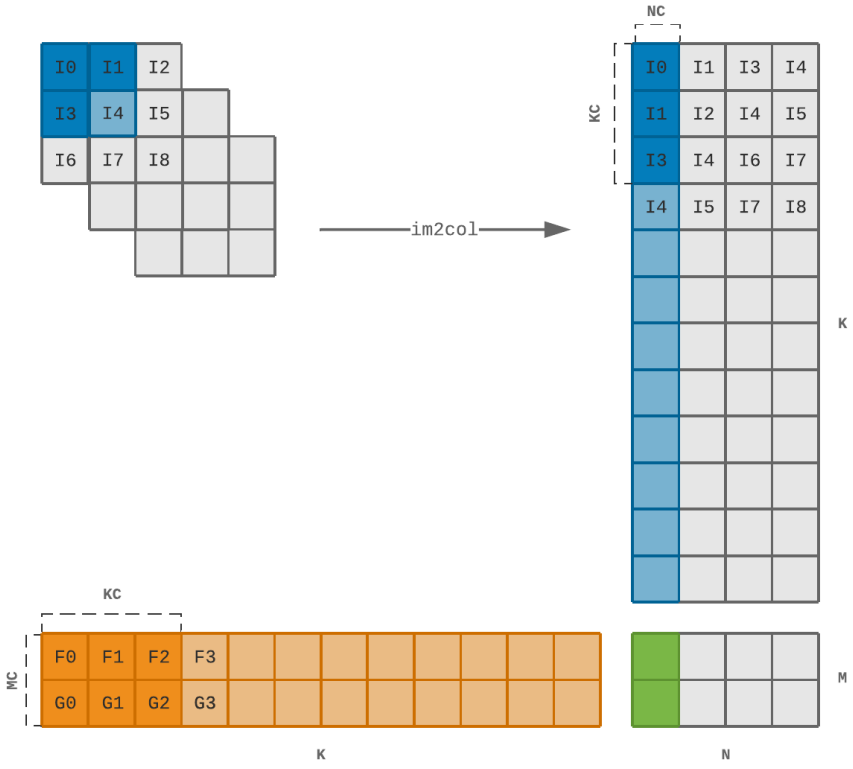

Figura 4.8: Exemplo co *buffer* empregado para almacenar as columnas da matriz intermedia necesaria no algoritmo GEMM-BLIS

Capítulo 5

Probas e resultados

Neste capítulo expoñeremos tanto o método seguido para a realización das probas como o resultado obtido nas mesmas. Comezaremos na sección 5.1 comentando as características do contorno de execución, Intel DevCloud, e incluíndo as especificacións dos distintos dispositivos dispoñibles nesta plataforma na nube. Deseguido, na sección 5.2, analizaremos os resultados da execución nas distintas plataformas, entre as que incluiremos tanto CPUs coma GPUs.

5.1 Contorno de probas

Como adiantamos nos primeiros capítulos desta memoria, para levar a cabo a execución das probas de rendemento fixemos uso dunha plataforma de computación na nube de Intel, coñecida como DevCloud[1]. Esta solución conta con tres modalidades:

- *Intel DevCloud for Edge* - Para aqueles desenvolvedores que traballen coa distribución OpenVINO de Intel, que ofrece redes de aprendizaxe profunda xa adestradas, co obxectivo de realizar probas na fase de inferencia.
- *Intel DevCloud for oneAPI* - Para desenvolvedores HPC que traballen coas ferramentas do *framework* Intel oneAPI. No noso caso, solicitamos acceso a DevCloud a través desta modalidade, que conta cun período de proba gratuíto de 120 días, ampliable baixo demanda.
- *Intel DevCloud for FPGA* - Para aqueles desenvolvedores especializados en dispositivos FPGA que precisen ferramentas máis específicas.

O seu funcionamento non difire do de calquera clúster de computación remota. En primeiro lugar, debemos conectarnos a un nodo *login* do clúster a través dun escritorio remoto

[1] https://software.intel.com/content/www/us/en/develop/tools/devcloud.html

ou SSH. Despois temos que transferir os nosos ficheiros fonte e compilalos. Finalmente, para executar os programas debemos enviar o traballo á cola a través dos comandos proporcionados polo programador de traballos do clúster.

No primeiro apartado desta sección, o 5.1.1, veremos as características físicas (*hardware*) dos dispositivos empregados na execución das probas, un dato fundamental para poder realizar comparacións igualitarias. Despois, no apartado 5.1.2, comentaremos o método e as ferramentas utilizadas para levar a cabo os experimentos.

5.1.1 Características dos dispositivos

En *Intel DevCloud for oneAPI* temos á nosa disposición unha selección de dispositivos que vai dende as CPUs ata as FPGAs, pasando polas GPUs. Tendo en conta que un dos obxectivos deste traballo era analizar o rendemento da nosa aplicación en plataformas heteroxéneas, decidimos executar as probas en dous dispositivos de diferente clase: un procesador *Intel Xeon E-2176G*[2] e a GPU integrada *Intel UHD Graphics P630 [0x3e96]*, cuxas características podemos ver resumidas na táboa 5.1. Cabe destacar que, ao non dispoñer de memoria gráfica dedicada, ambos dispositivos deben compartir a memoria principal. Ademais, o tamaño da memoria nos nodos de DevCloud que inclúen estes dispositivos é de 64 GB, cando os dispositivos poderían soportar ata 128 GB.

Procesador	**Intel® Xeon® E-2176G**
Frecuencia	3.70 GHz (4.70 GHz turbo)
Caché	12 MB Intel® Smart Cache
Número de núcleos	6 (ata 12 fíos)
Velocidade do bus	8 GT/s
Procesador gráfico	**Intel® UHD Graphics P630**
Frecuencia	0.35 - 1.20 GHz
Unidades de Execución (EU)	24
ID do dispositivo	0x3E96
Memoria	
Capacidade	64 GB (máx. 128GB)
Tipo	DDR4-2666 (dual-channel)
Ancho de banda	41.6 GB/s

Táboa 5.1: Características do procesador e a gráfica empregados na execución das probas

[2] https://ark.intel.com/content/www/us/en/ark/products/134860/intel-xeon-e-2176g-processor-12m-cache-up-to-4-70-ghz.html

```
#!/bin/bash
#PBS -N batch4
#PBS -l walltime=00:45:00
#PBS -l nodes=1:gpu:ppn=2
#PBS -d .

# Set the environment
source /opt/intel/inteloneapi/setvars.sh &> /dev/null

# Build the project
cd .. && ./build > /dev/null && cd bin/

# Run the tests
TIMEFORMAT='%4R';
echo "executable,device,parameters,time1,time2,time3,time4";

for executable in "direct" "gemm" "blis"; do
  for device in "cpu" "gpu"; do
    for params in\
      "8  4 4 1024 1024 3 3"\
      "16 4 4 1024 1024 3 3"\
      "32 4 4 1024 1024 3 3"\
      "64 4 4 1024 1024 3 3"
    do
      printf "${executable},${device},${params}"
      for i in {1..4}; do
        timei=$( { time ./${executable} ${device} ${params}; } 2>&1 )
        printf ",${timei}"
      done; echo
    done;
  done;
done;
```

Figura 5.1: *Script* de exemplo para a subscrición de traballos en DevCloud

5.1.2 Método de execución das probas

En primeiro lugar, á hora de escoller un método para realizar a medición do tempo de execución de cada algoritmo, decantámonos por un bastante sinxelo á par que eficaz. Este método baséase en implementar unha serie de programas clónicos que só difiran no algoritmo a probar, para despois medir o tempo de execución total dos programas mediante unha ferramenta externa, como pode ser o comando `time`. Como podemos deducir, nesta medición tamén estaremos incluíndo o tempo de execución daquelas operacións adicionais que sexa necesario realizar, como pode ser a copia dos datos do anfitrión ao dispositivo, no caso das execucións na GPU, ou a transformación dos datos ao formato requerido polo algoritmo. Desta forma tamén estamos obtendo unha medición máis realista, que nos ofrece unha visión global dos custos do problema.

En segundo lugar, para analizar a evolución do rendemento en función do tamaño do pro-

blema realizamos dous grupos de probas: o primeiro baséase en executar os distintos programas utilizando un número crecente de tensores de entrada, é dicir, aumentando o parámetro N en cada execución. O segundo aplica a mesma filosofía sobre as dimensións dos tensores, os parámetros H e W. Co obxectivo de automatizar a execución desta gran cantidade de probas, codificamos varios *scripts* listos para ser lanzados á cola de traballos do clúster. Podemos atopalos, xunto coa saída que xeraron, no directorio `job`[3] do repositorio. Como podemos observar na figura 5.1, deseñamos estes *scripts* de tal forma que a saída xerada polos distintos programas conforme un único ficheiro CSV, que despois poderá ser importado a unha folla de cálculo. Tamén podemos fixarnos en como, co obxectivo de evitar oscilacións na toma de tempos, repetimos cada proba ata en catro ocasións, para despois calcular a media aritmética.

Finalmente, cabe destacar que o ideal sería aproveitar a función de escritorio remoto de Intel DevCloud para facer unha análise do rendemento máis minuciosa coas ferramentas gráficas que ofrece o *framework* Intel oneAPI. Por exemplo, un seguinte paso podería ser utilizar as aplicacións estudadas na asignatura Ferramentas para HPC do mestrado, *Vtune Amplifier* e *Intel Inspector*, para mellorar a paralelización, a vectorización e, en definitiva, seguir as recomendacións que estas nos indiquen para axudar ao compilador a facer mellor o seu traballo.

5.2 Análise dos resultados

Nesta sección expoñeremos os resultados obtidos tras a execución das probas nos distintos dispositivos. Como adiantábamos en anteriores parágrafos, realizamos dous grupos de probas en función da variable a escalar: o número de imaxes no lote ou o tamaño das imaxes. Desta forma poderemos comprobar se hai situacións nas que destaque máis un algoritmo que outro.

Os programas aos que imos facer mención nos próximos apartados son: *direct_sequential* e *direct_parallel*, as nosas implementacións secuencial e paralela do algoritmo directo da convolución; *gemm_sequential* e *gemm_parallel*, as nosas implementacións secuencial e paralela do algoritmo GEMM; *blis_sequential* e *blis_parallel*, as nosas implementacións do algoritmo que bautizamos como GEMM-BLIS; e finalmente, *winograd_onednn*, *direct_onednn* e *gemm_onednn*, as distintas implementacións da convolución presentes na libraría oneDNN.

5.2.1 En función do número de imaxes no lote

A primeira das probas consistiu en realizar varias execucións de cada programa aumentando en cada unha delas o número de imaxes no lote, é dicir, o parámetro N (*batch size*). Este é o único parámetro cuxo valor varía, o resto son fixos. En concreto empregamos N imaxes de dimensións 4x1024x1024 (CxHxW) e $K = 4$ filtros de tamaño 4x3x3 (CxRxS). O tamaño escollido para o filtro é un dos máis habituais no adestramento de redes neuronais

[3] https://git.fic.udc.es/s.aguado/tfm/tree/master/job

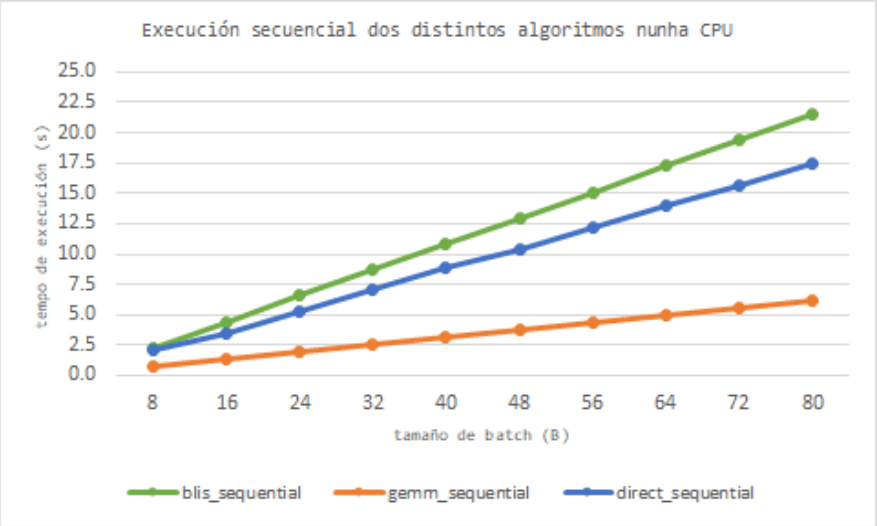

Figura 5.2: Tempo de execución na CPU, en función do número de imaxes no lote, dos programas *direct_sequential.cpp*, *gemm_sequential.cpp* e *blis_sequential.cpp*.

convolucionais, e ademais permitiunos comparar os nosos algoritmos cos xa implementados en oneDNN, pois algúns deles tiñan restricións en canto ás dimensións do mesmo.

En primeiro lugar, comezaremos analizando os resultados obtidos polos programas secuenciais: *direct_sequential*, *gemm_sequential* e *blis_sequential*. Na figura 5.2 podemos ver o tempo de execución dos programas citados en función do número de imaxes no lote de entrada, unha escala lineal de 8 a 80 imaxes. Como podemos observar no gráfico, o algoritmo *blis_sequential* obtén, contra as nosas expectativas, os peores resultados. Semella que o aproveitamento da memoria caché non foi suficiente para compensar o traballo extra que debe realizar o algoritmo. O que si puidemos prognosticar con éxito foi a obtención do primeiro posto por parte do algoritmo *gemm_sequential*. Trátase dun algoritmo moi simple cuxa rexión de interese, a multiplicación de matrices, é unha operación moi coñecida e a maior parte dos compiladores son capaces de optimizar o seu rendemento sen intervención do programador. Pola súa parte, o algoritmo *direct_sequential* serviranos como referencia á hora de avaliar a aceleración dos demais algoritmos.

En segundo lugar, realizamos a mesma comparación desta vez coas versións paralelas. Na figura 5.3 podemos ver o tempo de execución dos programas *direct_parallel*, *gemm_parallel* e *blis_parallel*, todos eles executados de forma paralela na CPU. O primeiro que vemos a simple vista é que o tempo de execución, en xeral, vese moi reducido: cando antes investíamos preto de 20 segundos no procesamento de 80 imaxes, na versión paralela non superamos os 5 segundos, o que supón unha aceleración de factor $x4$. O segundo punto a destacar é a gran mellora do algoritmo *blis_parallel* que, grazas ás modificacións introducidas no algoritmo orixinal, iguala en rendemento ao seu predecesor -o algoritmo *gemm_parallel*- á vez que engade unha

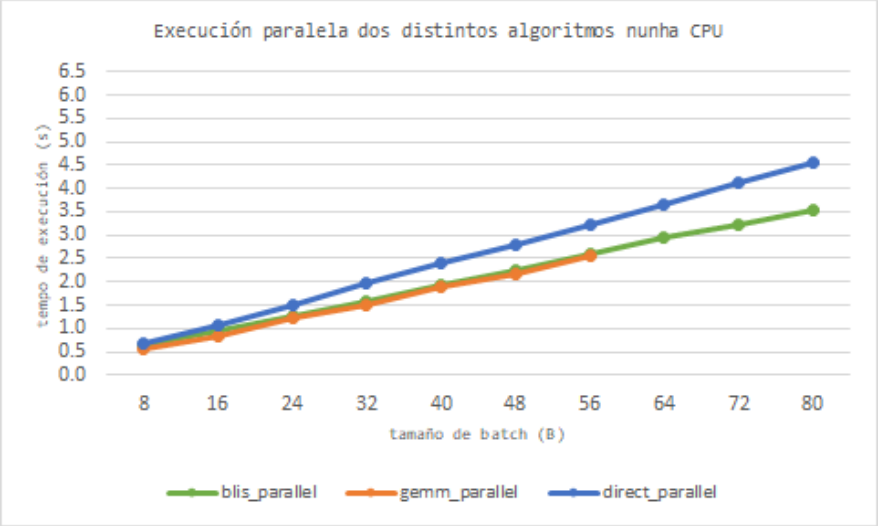

Figura 5.3: Tempo de execución na CPU, en función do número de imaxes no lote, dos programas *direct_parallel.cpp*, *gemm_parallel.cpp* e *blis_parallel.cpp*.

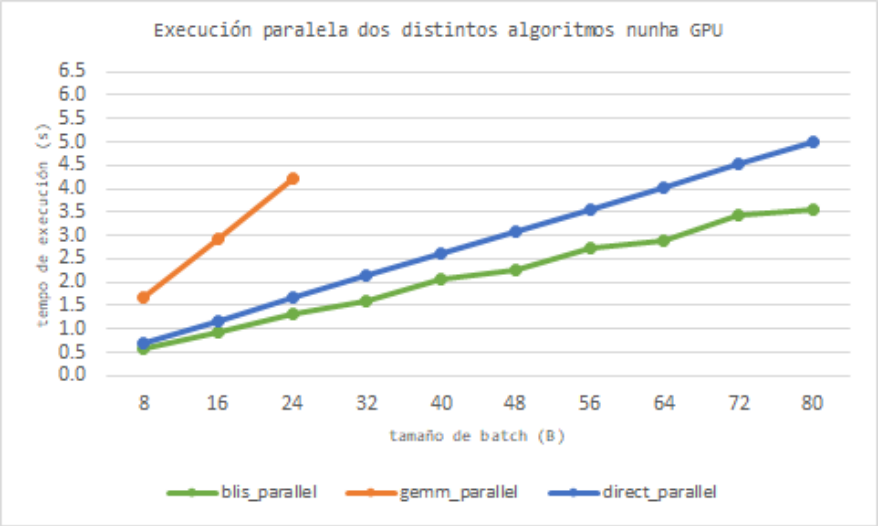

Figura 5.4: Tempo de execución na GPU, en función do número de imaxes no lote, dos programas *direct_parallel.cpp*, *gemm_parallel.cpp* e *blis_parallel.cpp*.

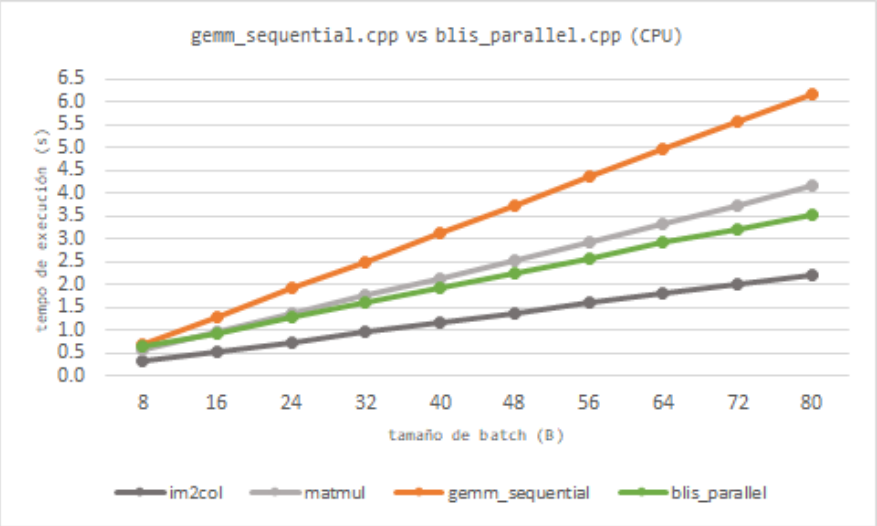

Figura 5.5: Tempo de execución na CPU, en función do número de imaxes no lote, do algoritmo *gemm_sequential*, das funcións que o conforman: *im2col* e *matmul*; e do algoritmo *blis_parallel*.

maior escalabilidade. Ligamos así co terceiro e último punto pois, como podemos observar no gráfico, para o algoritmo *gemm_parallel* non temos datos máis aló das 56 imaxes. Isto é debido a que os requerimentos de memoria do algoritmo superan os 64GB dispoñibles no nodo de cómputo, o que supón un gran problema en aplicacións do mundo real. Por outro lado, na figura 5.4 representamos o tempo de execución dos mesmos programas, desta vez, executados de forma paralela nunha GPU. Como vemos, o rendemento dos algoritmos *direct_parallel* e *blis_parallel* non cambia demasiado, pero si que vemos unha clara diferencia no algoritmo *gemm_parallel*. Se lembramos a implementación deste programa, a necesidade dunha sincronización entre as operacións que o conforman obrigábanos a implementar dous *kernels* e a utilizar a instrución `wait` entre ambos. No caso anterior, cando executábamos o programa na CPU, a sincronización non supoñía ningún trastorno, pois o dispositivo era o propio anfitrión e utilizaba as mesmas posicións de memoria. En cambio agora, cando executamos o programa na GPU e invocamos á función `wait` despois do primeiro *kernel*, o *runtime* ten a obriga de sincronizar ambos dispositivos, tamén a nivel de memoria, o que supón un gran sobrecusto que torna inviable unha vez máis a escolla deste algoritmo.

En terceiro lugar, imos comparar os algoritmos *gemm_sequential* e *blis_parallel*, ambos executados na CPU, co obxectivo de ver graficamente a mellora que introduce este último. Na figura 5.5 temos os resultados destes dous programas, xunto cos resultados individuais das funcións `im2col` e `matmul`. Como podemos ver no gráfico, co algoritmo *blis_parallel* somos capaces de aforrar o custo de realizar a operación `im2col`, ademais de gañar en escalabilidade, como xa comentamos anteriormente.

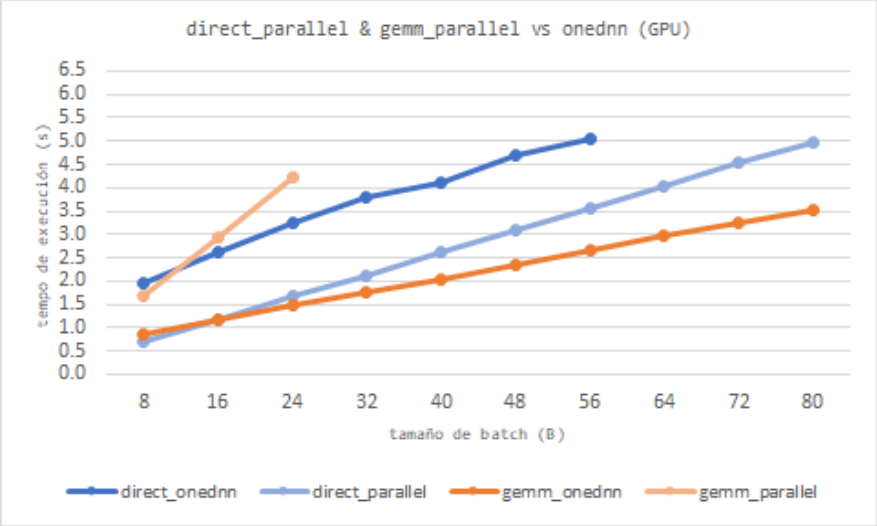

Figura 5.6: Tempo de execución na GPU, en función do número de imaxes no lote, dos algoritmos soportados pola libraría oneDNN: *direct_onednn* e *gemm_onednn*; fronte aos nosos algoritmos *direct_parallel* e *gemm_parallel*.

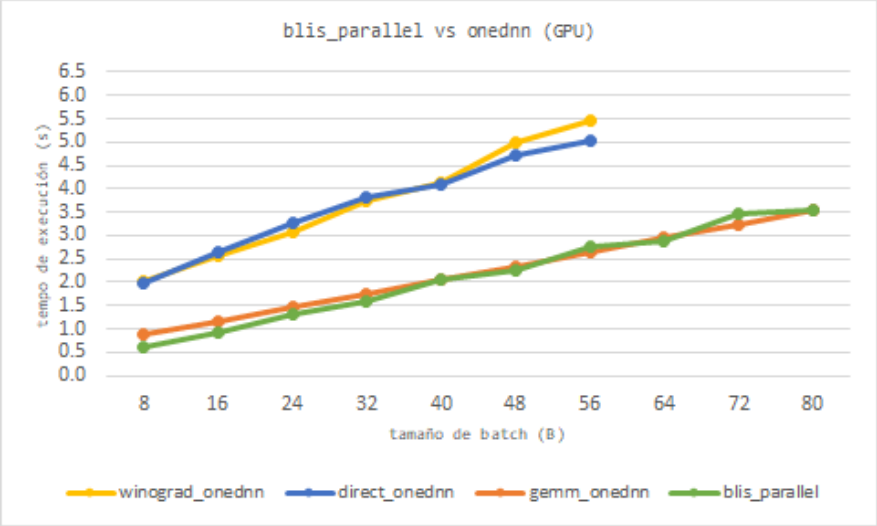

Figura 5.7: Tempo de execución na GPU, en función do número de imaxes no lote, dos algoritmos soportados pola libraría oneDNN: *direct_onednn*, *gemm_onednn* e *winograd_onednn*; e do noso algoritmo *blis_parallel*.

En cuarto lugar, nas figuras 5.6 e 5.7 comparamos o tempo de execución dos algoritmos soportados pola libraría oneDNN (*winograd_onednn*, *direct_onednn* e *gemm_onednn*) fronte ás nosas implementacións. Comezamos na 5.6 cos algoritmos *direct_parallel* e *gemm_parallel*. Como vemos, o noso algoritmo directo (*direct_parallel*) consome menos tempo que o da libraría (*direct_onednn*), pois non necesita facer ningunha transformación sobre o formato dos datos. En cambio, co algoritmo *gemm_parallel* obtemos un rendemento e unha escalabilidade moito peores que co *gemm_onednn*. Como este último non presenta problemas de espazo, ao contrario que a nosa implementación do algoritmo GEMM, intuímos que debe estar empregando unha versión optimizada, similar á que nós bautizamos como GEMM-BLIS. Por este motivo, na figura 5.7 comparamos o noso algoritmo *blis_parallel* fronte ao algoritmo GEMM da libraría, *gemm_onednn*, demostrando así que a nosa implementación ofrece uns resultados moi similares. Ademais, na mesma figura incluímos os outros dous algoritmos implementados por oneDNN: *direct_onednn* e *winograd_onednn*; sendo estes dous os que máis tempo precisan dos catro, pois se ven obrigados a transformar os datos de entrada do formato no que chegan ao formato que necesitan as primitivas.

Xa por último, no anexo A.1 deixamos unha táboa coa media das catro execucións realizadas para cada algoritmo, e a súa aceleración fronte o programa *direct_sequential*.

5.2.2 En función do tamaño das imaxes

O segundo grupo de probas baséase en realizar varias execucións de cada programa aumentando en cada unha delas o tamaño das imaxes utilizadas, é dicir, os parámetros H e W (altura e largura das imaxes); mentres o resto de valores permanecen fixos. En concreto empregamos $N = 8$ imaxes con $C = 4$ mapas de características (ou canles) e $K = 4$ filtros de dimensións 4x3x3 (CxRxS). Unha configuración que non difire demasiado da escollida para as probas do apartado anterior.

Unha vez máis, imos comezar analizando os resultados obtidos pola versión secuencial dos tres algoritmos implementados (*direct_sequential*, *gemm_sequential* e *blis_sequential*). Representámolos graficamente na figura 5.8, na que podemos ver como o tempo de execución aumenta de forma lineal co tamaño das entradas. Cabe salientar que o tamaño das imaxes utilizadas nas probas aumenta en potencias de dous, de 64x64 a 4096x4096, motivo polo cal vemos a gráfica como unha curva. Se nos fixamos nela, de novo temos que o algoritmo máis lento é *blis_sequential* mentres que o máis rápido segue sendo *gemm_sequential*.

En segundo lugar, comparamos o resultado dos algoritmos paralelos (é dicir, os programas *direct_parallel*, *gemm_parallel* e *blis_parallel*) executados tanto na CPU (figura 5.9) como na GPU (figura 5.10). Por un lado, con estas versións volvemos experimentar unha baixada drástica do tempo de execución, que pasa de 35 a menos de oito segundos, acadando no mellor caso unha aceleración de factor x5. Tamén para este caso de proba, o algoritmo *blis_parallel*

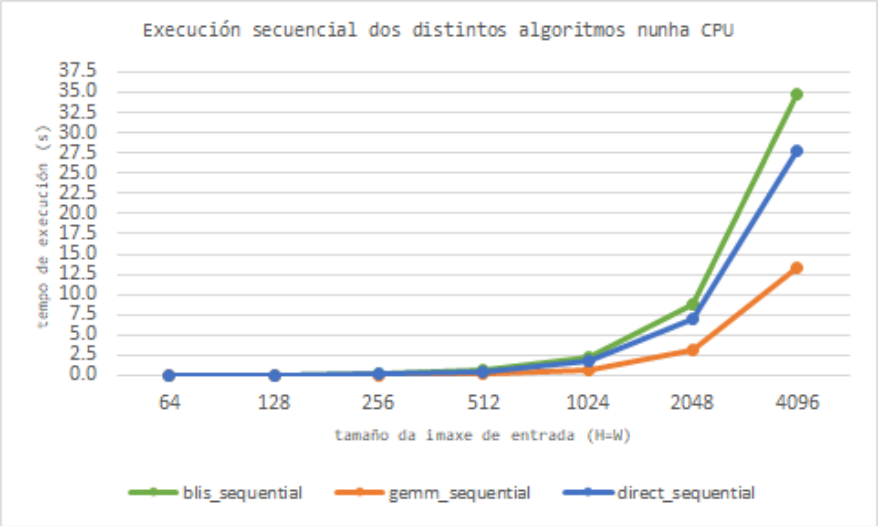

Figura 5.8: Tempo de execución na CPU, en función do tamaño das imaxes, dos programas *direct_sequential.cpp*, *gemm_sequential.cpp* e *blis_sequential.cpp*.

volve ser o máis veloz. Por outro lado, se comparamos as dúas figuras, podemos notar unha baixada no rendemento do algoritmo *gemm_parallel* cando este se executa nunha GPU. Como comentábamos no apartado anterior, isto é debido a problemas derivados da sincronización, mais neste caso o cambio apreciado non é tan drástico. Finalmente, segue sendo evidente o problema de escalabilidade do algoritmo *gemm_parallel* que, debido á gran cantidade de memoria que precisa, non pode ser executado coa totalidade de tamaños de entrada.

En terceiro lugar, na figura 5.11 mostramos unha comparativa entre as gráficas do algoritmo *gemm_sequential* (e as funcións nas que se descompón) fronte ao algoritmo *blis_parallel*, obtendo uns resultados moi similares aos do apartado anterior.

Finalizamos o capítulo cunha rápida visita ás dúas últimas gráficas. Na primeira, a figura 5.12, comparamos os nosos algoritmos *direct_parallel* e *gemm_parallel* cos da libraría: *direct_onednn* e *gemm_onednn*. En resumidas contas, o programa *direct_parallel* volve ser máis rápido que o seu homólogo da libraría, mentres que o *gemm_parallel* perde, unha vez máis, contra a escalabilidade de oneDNN. Na segunda gráfica, a figura 5.13, comparamos os algoritmos soportados pola libraría oneDNN coa nosa mellor implementación, a do programa *blis_parallel*. Como podemos observar, o noso programa ofrece un rendemento equivalente ao da libraría, sendo lixeiramente superior con tamaños de entrada relativamente pequenos.

En conclusión, grazas a este segundo grupo de probas asegurámonos de que o algoritmo *blis_parallel* segue sendo o que mellores resultados ofrece, aínda que as necesidades do problema varíen. No anexo A.2 deixamos unha táboa coa media das catro execucións realizadas para cada algoritmo neste grupo de probas, e a súa aceleración fronte a versión directa e secuencial da convolución.

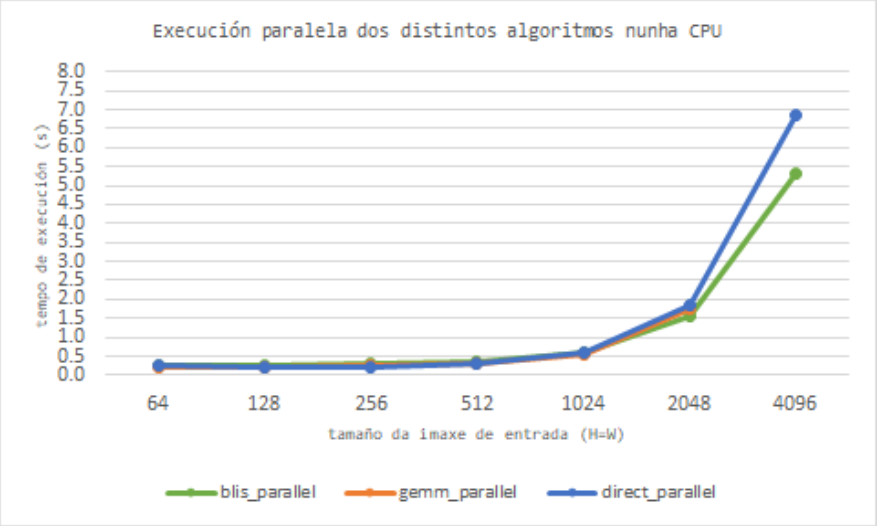

Figura 5.9: Tempo de execución na CPU, en función do tamaño das imaxes, dos programas *direct_parallel.cpp*, *gemm_parallel.cpp* e *blis_parallel.cpp*.

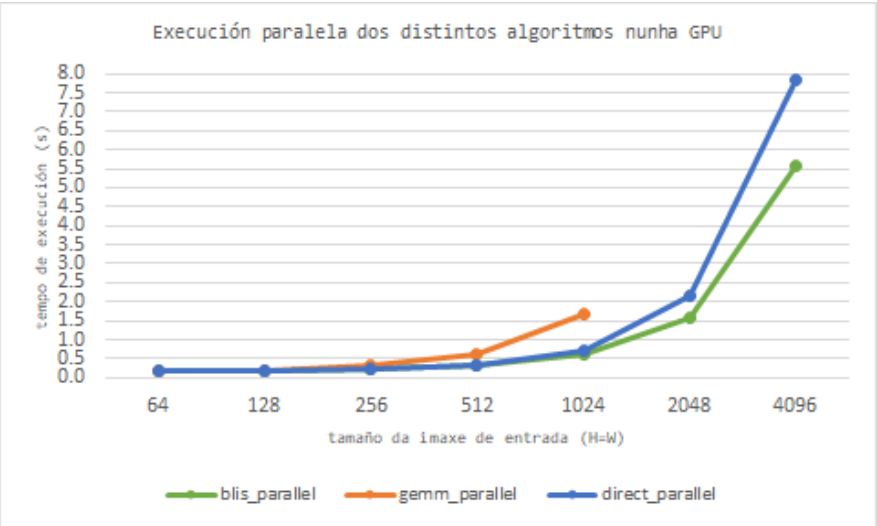

Figura 5.10: Tempo de execución na GPU, en función do tamaño das imaxes, dos programas *direct_parallel.cpp*, *gemm_parallel.cpp* e *blis_parallel.cpp*.

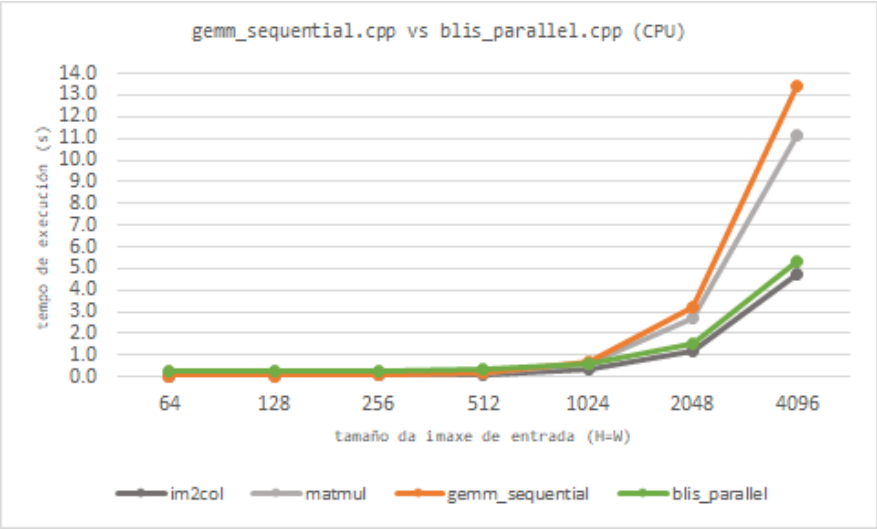

Figura 5.11: Tempo de execución na CPU, en función do tamaño das imaxes, do algoritmo *gemm_sequential*, das funcións que o conforman: *im2col* e *matmul*; e do algoritmo *blis_parallel*.

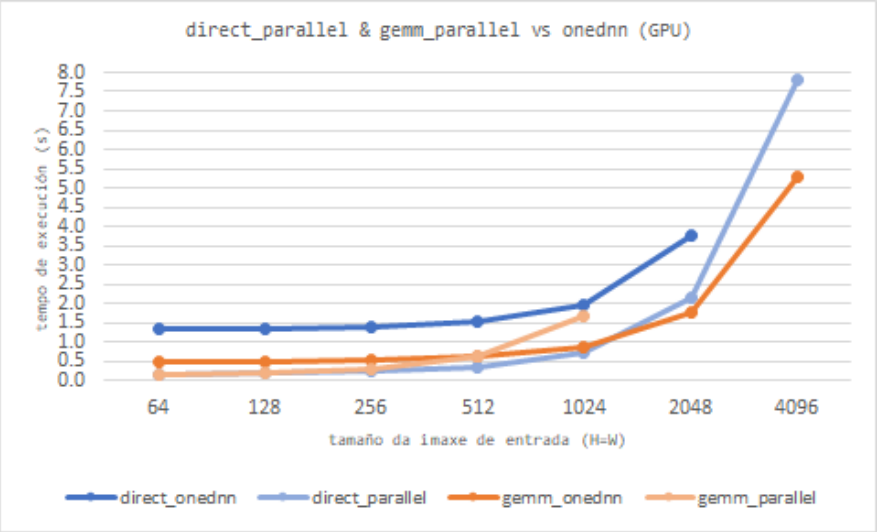

Figura 5.12: Tempo de execución na GPU, en función do tamaño das imaxes, dos algoritmos soportados pola libraría oneDNN: *direct_onednn* e *gemm_onednn*; fronte aos nosos algoritmos *direct_parallel* e *gemm_parallel*.

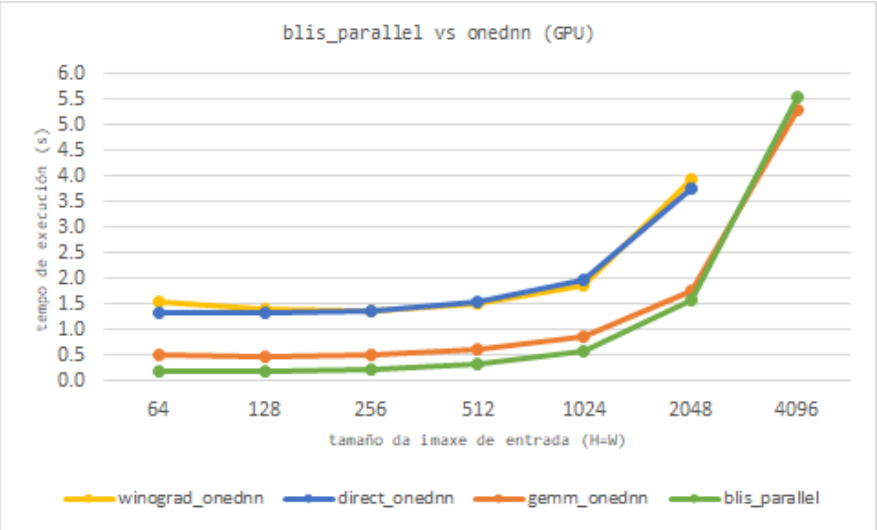

Figura 5.13: Tempo de execución na GPU, en función do tamaño das imaxes, dos algoritmos soportados pola libraría oneDNN: *direct_onednn*, *gemm_onednn* e *winograd_onednn*; e do noso algoritmo *blis_parallel*.

Capítulo 6

Conclusións

Neste derradeiro capítulo, realizaremos un repaso do traballo desenvolvido durante o proxecto. Resumindo a situación final do mesmo e comentando a súa concordancia cos obxectivos definidos inicialmente.

En primeiro lugar, na sección 6.1 comezaremos co repaso do traballo realizado e o contraste de obxectivos. En segundo lugar, na sección 6.2 mencionaremos aquelas competencias do mestrado que gardan relación con este traballo. Finalmente, na sección 6.3 faremos unha breve mención ás tarefas que quedaron pendentes ou ás posibles liñas futuras.

6.1 Recapitulación

A continuación, nesta sección realizaremos un breve resumo do traballo realizado nos distintos capítulos da memoria.

En primeiro lugar, no capítulo 1 expuxemos os obxectivos do proxecto, consistindo o principal en estudar o *framework* oneAPI para despois implementar un algoritmo nesta plataforma. O resultado da investigación, os fundamentos da operación e a ferramenta, foron plasmados no capítulo 2. Unha vez rematado este traballo previo, dispuxémonos a comezar coas tarefas que recollemos na planificación do capítulo 3. No capítulo 4 describimos os detalles de codificación dos algoritmos implementados, cumprindo así co segundo dos obxectivos. E finalmente, no capítulo 5 realizamos a análise do rendemento dos nosos algoritmos fronte a implementacións xa existentes, tal e como contemplábamos no último. Podemos considerar, polo tanto, que cumprimos con todos os obxectivos que definimos inicialmente.

6.2 Competencias da titulación

Se ben é certo que no *Mestrado en Computación de Altas Prestacións / High Performance Computing* cursamos unha materia, *Programación de Arquitecturas Heteroxéneas*, na que es-

tudamos a programación en *frameworks* orientados a plataformas heteroxéneas, como poden ser CUDA ou OpenCL, a realidade é que non houbo ningunha aula sobre a linguaxe DPC++. Podemos considerar polo tanto, que o estudo levado a cabo neste traballo complementa as ensinanzas do mestrado. Por outro lado, na materia *Ferramentas para HPC*, traballamos con ferramentas dispoñibles no *framework* Intel oneAPI, mais sobre códigos preexistentes.

A continuación, imos facer mención a unha relación de competencias propias da titulación que coinciden coas tarefas ou conceptos levados á práctica neste proxecto.

- CE4 - *Afondar no coñecemento de ferramentas de programación e diferentes linguaxes no campo da computación de altas prestacións*

  Un dos obxectivos deste proxecto era estudar o *framework* Intel oneAPI e, máis concretamente, a linguaxe de programación DPC++, unha ferramenta especialmente deseñada para implementar programas paralelos orientados a arquitecturas HPC.

- CE9 - *Ser capaz de formular, modelar e resolver problemas que necesitan de técnicas de computación de altas prestacións*

  Como non partimos de ningún código existente, tivemos que elaborar todo o proxecto: dende a planificación, pasando pola busca de fontes de información, ata o deseño e implementación dos algoritmos.

- CE5 - *Analizar, deseñar e implementar algoritmos e aplicacións paralelas eficientes*

  O principal obxectivo deste proxecto, e que podemos considerar cumprido, era implementar un algoritmo paralelo para optimizar a operación da convolución por lotes.

- CE2 - *Analizar e mellorar o rendemento dunha arquitectura ou un software dado*

  Outro dos obxectivos inicialmente formulados era avaliar o rendemento do noso algoritmo e o das implementacións xa existentes. Ademais, apoiámonos nas análises realizadas para iterar sobre os algoritmos mellorando o seu rendemento.

- CT3 - *Capacidade de xestionar tempos e recursos: desenvolver plans, priorizar actividades, identificar as críticas, establecer prazos e cumprilos*

  Antes de comezar o proxecto realizamos unha planificación onde definimos as tarefas a realizar e a súa programación sobre o calendario. Á hora de estimar o esforzo de cada tarefa procuramos ser realistas, grazas a iso agora podemos afirmar que foi posible completar todas as tarefas no tempo proposto inicialmente.

6.3 Liñas futuras

Aínda tendo cumprido cos prazos e os obxectivos definidos inicialmente, o certo é que se nos ocorreron certas melloras que, por falta de tempo, non puidemos levar á práctica.

A primeira delas comentámola no capítulo 5, e consistiría en utilizar o escritorio remoto de Intel DevCloud para analizar os nosos programas a través das ferramentas do propio *framework*, Intel oneAPI, como poden ser Intel Advisor ou Intel VTune. Grazas a esta análise poderíamos refinar a nosa paralelización ou incluso as opcións de compilación empregadas, co obxectivo de mellorar o rendemento final das aplicacións.

A segunda mellora ten que ver con afondar no estudo da linguaxe DPC++. Por exemplo, poderíamos utilizar as opcións de comunicación e sincronización dos elementos de procesamento que proporciona a linguaxe para intentar mellorar o algoritmo GEMM, cuxo rendemento víase moi afectado pola necesidade de sincronización entre as dúas funcións que o conforman.

Finalmente, outra posible liña futura consistiría en empregar algunha libraría especializada en operacións de álxebra lineal, como pode ser Intel oneMKL, para optimizar a multiplicación de matrices utilizada nos algoritmos GEMM e BLIS. Este tipo de librarías utilizan técnicas que aproveitan a localidade dos datos, por exemplo o *loop tiling*, para optimizar os patróns de acceso a memoria e acelerar as transaccións. Outra técnica moi coñecida, o *loop unrolling*, permite reducir o número de instrucións de salto que ten que realizar o procesador, reducindo ao mesmo tempo a súa carga de traballo.

Apéndices

Apéndice A

Táboas de resultados

A.1 Tempo de execución medio e aceleración (*speedup*) dos programas escalando a variable N

Algoritmo	Dispositivo	N	Media (s)	Aceleración
direct_sequential	cpu	8	2.100	1.000
direct_sequential	cpu	16	3.520	1.000
direct_sequential	cpu	24	5.263	1.000
direct_sequential	cpu	32	7.096	1.000
direct_sequential	cpu	40	8.841	1.000
direct_sequential	cpu	48	10.448	1.000
direct_sequential	cpu	56	12.189	1.000
direct_sequential	cpu	64	13.927	1.000
direct_sequential	cpu	72	15.660	1.000
direct_sequential	cpu	80	17.388	1.000
gemm_sequential	cpu	8	0.699	3.006
gemm_sequential	cpu	16	1.295	2.718
gemm_sequential	cpu	24	1.924	2.736
gemm_sequential	cpu	32	2.502	2.836
gemm_sequential	cpu	40	3.139	2.817
gemm_sequential	cpu	48	3.747	2.789
gemm_sequential	cpu	56	4.351	2.801
gemm_sequential	cpu	64	4.992	2.790
gemm_sequential	cpu	72	5.574	2.809
gemm_sequential	cpu	80	6.189	2.810

blis_sequential	cpu	8	2.205	0.952
blis_sequential	cpu	16	4.365	0.806
blis_sequential	cpu	24	6.570	0.801
blis_sequential	cpu	32	8.686	0.817
blis_sequential	cpu	40	10.818	0.817
blis_sequential	cpu	48	12.961	0.806
blis_sequential	cpu	56	15.102	0.807
blis_sequential	cpu	64	17.302	0.805
blis_sequential	cpu	72	19.469	0.804
blis_sequential	cpu	80	21.590	0.805
direct_parallel	cpu	8	0.680	3.089
direct_parallel	cpu	16	1.069	3.292
direct_parallel	cpu	24	1.515	3.474
direct_parallel	cpu	32	1.957	3.627
direct_parallel	cpu	40	2.390	3.699
direct_parallel	cpu	48	2.804	3.727
direct_parallel	cpu	56	3.241	3.761
direct_parallel	cpu	64	3.671	3.794
direct_parallel	cpu	72	4.140	3.783
direct_parallel	cpu	80	4.536	3.833
direct_parallel	gpu	8	0.707	2.972
direct_parallel	gpu	16	1.176	2.992
direct_parallel	gpu	24	1.679	3.135
direct_parallel	gpu	32	2.127	3.337
direct_parallel	gpu	40	2.619	3.376
direct_parallel	gpu	48	3.093	3.378
direct_parallel	gpu	56	3.571	3.413
direct_parallel	gpu	64	4.022	3.463
direct_parallel	gpu	72	4.520	3.464
direct_parallel	gpu	80	4.984	3.488
gemm_parallel	cpu	8	0.547	3.837
gemm_parallel	cpu	16	0.855	4.117
gemm_parallel	cpu	24	1.208	4.355
gemm_parallel	cpu	32	1.499	4.732
gemm_parallel	cpu	40	1.886	4.688

gemm_parallel	cpu	48	2.184	4.783
gemm_parallel	cpu	56	2.567	4.748
gemm_parallel	cpu	64	[1]	
gemm_parallel	cpu	72		
gemm_parallel	cpu	80		
gemm_parallel	gpu	8	1.666	1.260
gemm_parallel	gpu	16	2.914	1.208
gemm_parallel	gpu	24	4.226	1.245
gemm_parallel	gpu	32		
gemm_parallel	gpu	40		
gemm_parallel	gpu	48		
gemm_parallel	gpu	56		
gemm_parallel	gpu	64		
gemm_parallel	gpu	72		
gemm_parallel	gpu	80		
blis_parallel	cpu	8	0.629	3.340
blis_parallel	cpu	16	0.945	3.727
blis_parallel	cpu	24	1.281	4.107
blis_parallel	cpu	32	1.598	4.440
blis_parallel	cpu	40	1.914	4.618
blis_parallel	cpu	48	2.247	4.650
blis_parallel	cpu	56	2.580	4.724
blis_parallel	cpu	64	2.941	4.735
blis_parallel	cpu	72	3.223	4.858
blis_parallel	cpu	80	3.547	4.902
blis_parallel	gpu	8	0.592	3.548
blis_parallel	gpu	16	0.936	3.760
blis_parallel	gpu	24	1.312	4.012
blis_parallel	gpu	32	1.587	4.471
blis_parallel	gpu	40	2.047	4.320
blis_parallel	gpu	48	2.253	4.637
blis_parallel	gpu	56	2.742	4.445
blis_parallel	gpu	64	2.878	4.839
blis_parallel	gpu	72	3.453	4.536

[1] Os espazos en branco indican que por limitacións de memoria dita proba non puido ser executada.

blis_parallel	gpu	80	3.555	4.891
direct_onednn	cpu	8	0.850	2.4717
direct_onednn	cpu	16	0.698	5.043
direct_onednn	cpu	24	0.980	5.369
direct_onednn	cpu	32	1.226	5.790
direct_onednn	cpu	40	1.538	5.749
direct_onednn	cpu	48	1.789	5.839
direct_onednn	cpu	56	2.059	5.919
direct_onednn	cpu	64	2.346	5.936
direct_onednn	cpu	72	2.609	6.003
direct_onednn	cpu	80	2.890	6.017
direct_onednn	gpu	8	1.974	1.064
direct_onednn	gpu	16	2.634	1.336
direct_onednn	gpu	24	3.257	1.616
direct_onednn	gpu	32	3.801	1.867
direct_onednn	gpu	40	4.108	2.152
direct_onednn	gpu	48	4.710	2.219
direct_onednn	gpu	56	5.029	2.424
direct_onednn	gpu	64		
direct_onednn	gpu	72		
direct_onednn	gpu	80		
gemm_onednn	cpu	8	0.484	4.338
gemm_onednn	cpu	16	0.771	4.565
gemm_onednn	cpu	24	1.055	4.991
gemm_onednn	cpu	32	1.361	5.216
gemm_onednn	cpu	40	1.641	5.389
gemm_onednn	cpu	48	1.950	5.357
gemm_onednn	cpu	56	2.236	5.450
gemm_onednn	cpu	64	2.530	5.505
gemm_onednn	cpu	72	2.819	5.555
gemm_onednn	cpu	80	3.123	5.568
gemm_onednn	gpu	8	0.864	2.430
gemm_onednn	gpu	16	1.154	3.049
gemm_onednn	gpu	24	1.473	3.574
gemm_onednn	gpu	32	1.753	4.047

gemm_onednn	gpu	40	2.052	4.309
gemm_onednn	gpu	48	2.348	4.450
gemm_onednn	gpu	56	2.661	4.580
gemm_onednn	gpu	64	2.954	4.714
gemm_onednn	gpu	72	3.235	4.842
gemm_onednn	gpu	80	3.524	4.934
winograd_onednn	gpu	8	2.001	1.050
winograd_onednn	gpu	16	2.583	1.363
winograd_onednn	gpu	24	3.093	1.702
winograd_onednn	gpu	32	3.740	1.897
winograd_onednn	gpu	40	4.127	2.142
winograd_onednn	gpu	48	4.988	2.095
winograd_onednn	gpu	56	5.468	2.229
winograd_onednn	gpu	64		
winograd_onednn	gpu	72		
winograd_onednn	gpu	80		

A.2 Tempo de execución medio e aceleración (*speedup*) dos programas escalando as variables H e W

Algoritmo	Dispositivo	H/W	Media (s)	Aceleración
direct_sequential	cpu	64	0.075	1.000
direct_sequential	cpu	128	0.072	1.000
direct_sequential	cpu	256	0.157	1.000
direct_sequential	cpu	512	0.483	1.000
direct_sequential	cpu	1024	1.784	1.000
direct_sequential	cpu	2048	7.013	1.000
direct_sequential	cpu	4096	27.872	1.000
gemm_sequential	cpu	64	0.050	1.513
gemm_sequential	cpu	128	0.053	1.381
gemm_sequential	cpu	256	0.079	1.994
gemm_sequential	cpu	512	0.210	2.298
gemm_sequential	cpu	1024	0.697	2.558
gemm_sequential	cpu	2048	3.233	2.169
gemm_sequential	cpu	4096	13.423	2.076
blis_sequential	cpu	64	0.055	1.362
blis_sequential	cpu	128	0.080	0.912
blis_sequential	cpu	256	0.181	0.863
blis_sequential	cpu	512	0.586	0.825
blis_sequential	cpu	1024	2.204	0.809
blis_sequential	cpu	2048	8.705	0.806
blis_sequential	cpu	4096	34.672	0.804
direct_parallel	cpu	64	0.244	0.309
direct_parallel	cpu	128	0.210	0.345
direct_parallel	cpu	256	0.228	0.686
direct_parallel	cpu	512	0.308	1.568
direct_parallel	cpu	1024	0.617	2.891
direct_parallel	cpu	2048	1.828	3.836
direct_parallel	cpu	4096	6.845	4.072
direct_parallel	gpu	64	0.158	0.477
direct_parallel	gpu	128	0.180	0.404
direct_parallel	gpu	256	0.224	0.699

direct_parallel	gpu	512	0.347	1.392
direct_parallel	gpu	1024	0.712	2.504
direct_parallel	gpu	2048	2.141	3.276
direct_parallel	gpu	4096	7.819	3.565
gemm_parallel	cpu	64	0.220	0.342
gemm_parallel	cpu	128	0.219	0.332
gemm_parallel	cpu	256	0.235	0.666
gemm_parallel	cpu	512	0.291	1.659
gemm_parallel	cpu	1024	0.538	3.314
gemm_parallel	cpu	2048	1.765	3.972
gemm_parallel	cpu	4096		
gemm_parallel	gpu	64	0.170	0.443
gemm_parallel	gpu	128	0.198	0.366
gemm_parallel	gpu	256	0.312	0.502
gemm_parallel	gpu	512	0.629	0.768
gemm_parallel	gpu	1024	1.665	1.078
gemm_parallel	gpu	2048		
gemm_parallel	gpu	4096		
blis_parallel	cpu	64	0.278	0.271
blis_parallel	cpu	128	0.278	0.261
blis_parallel	cpu	256	0.289	0.541
blis_parallel	cpu	512	0.355	1.359
blis_parallel	cpu	1024	0.601	2.966
blis_parallel	cpu	2048	1.537	4.562
blis_parallel	cpu	4096	5.324	5.235
blis_parallel	gpu	64	0.197	0.383
blis_parallel	gpu	128	0.197	0.368
blis_parallel	gpu	256	0.230	0.680
blis_parallel	gpu	512	0.325	1.486
blis_parallel	gpu	1024	0.591	3.018
blis_parallel	gpu	2048	1.587	4.420
blis_parallel	gpu	4096	5.559	5.014
direct_onednn	cpu	64	0.600	0.125
direct_onednn	cpu	128	0.147	0.494
direct_onednn	cpu	256	0.154	1.018

direct_onednn	cpu	512	0.207	2.335
direct_onednn	cpu	1024	0.411	4.337
direct_onednn	cpu	2048	1.234	5.683
direct_onednn	cpu	4096	4.488	6.210
direct_onednn	gpu	64	1.329	0.057
direct_onednn	gpu	128	1.347	0.054
direct_onednn	gpu	256	1.367	0.115
direct_onednn	gpu	512	1.539	0.314
direct_onednn	gpu	1024	1.964	0.908
direct_onednn	gpu	2048	3.757	1.867
direct_onednn	gpu	4096		
gemm_onednn	cpu	64	0.168	0.447
gemm_onednn	cpu	128	0.153	0.473
gemm_onednn	cpu	256	0.166	0.943
gemm_onednn	cpu	512	0.221	2.187
gemm_onednn	cpu	1024	0.437	4.082
gemm_onednn	cpu	2048	1.305	5.374
gemm_onednn	cpu	4096	77.255	0.361
gemm_onednn	gpu	64	0.503	0.149
gemm_onednn	gpu	128	0.495	0.146
gemm_onednn	gpu	256	0.521	0.301
gemm_onednn	gpu	512	0.610	0.792
gemm_onednn	gpu	1024	0.864	2.064
gemm_onednn	gpu	2048	1.760	3.985
gemm_onednn	gpu	4096	5.289	5.269
winograd_onednn	gpu	64	1.553	0.048
winograd_onednn	gpu	128	1.393	0.052
winograd_onednn	gpu	256	1.382	0.113
winograd_onednn	gpu	512	1.505	0.321
winograd_onednn	gpu	1024	1.863	0.957
winograd_onednn	gpu	2048	3.931	1.784
winograd_onednn	gpu	4096		

Apéndice B

Relación de acrónimos

ALU *Arithmetic Logic Unit*

ANN *Artificial Neural Network*

API *Application Programming Interface*

BLAS *Basic Linear Algebra Subprograms*

BLIS *BLAS-like Library Instantation Software*

CNN *Convolutional Neural Network*

CPU *Central Processing Unit*

CUDA *Compute Unified Device Architecture*

DBN *Deep Belief Network*

DNN *Deep Neural Network*

DPC++ *Data Parallel C++*

DSP *Digital Signal Processor*

FC *Fully Connected*

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*

OpenCL *Open Computing Language*

OpenGL *Open Graphics Library*

PC *Personal Computer*

SIMD *Single Instruction Multiple Data*

RBM *Restricted Boltzmann Machines*

ReLU *Rectified Linear Units*

RNN *Recurrent Neural Network*

UPN *Unsupervised Pretrained Network*

USM *Unified Shared Memory*

Apéndice C

Glosario de termos

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.

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

Clúster 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, incluído no *Machine Learning*, 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, como poden ser librarías ou linguaxes de programación, que serve de base para o desenvolvemento de software.

Host Dispositivo anfitrión, xeralmente unha CPU, en contraposición ao dispositivo invitado, por exemplo unha GPU.

Kernel Función executada de forma concorrente nun dispositivo.

Machine Learning Campo da intelixencia artificial que estuda o desenvolvemento de algoritmos que melloran en base da experiencia e os datos.

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.

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

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

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

Bibliografía

[1] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, and X. Tian, *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. Berkeley, CA: Apress, 2021, Data de consulta: 23 de xuño de 2021. [Online]. Available: https://doi.org/10.1007/978-1-4842-5574-2_1

[2] S. A. Couselo and D. A. Canosa, *Implementación en CUDA dun método para realizar a operación de convolución en lotes*. Universidade da Coruña, 2020, Data de consulta: 23 de xuño de 2021. Dispoñible en: https://hdl.handle.net/2183/26686

[3] Intel, "oneAPI Specification," Data de consulta: 23 de xuño de 2021. Dispoñible en: https://docs.oneapi.com/versions/latest/index.html

[4] ——, "oneAPI Deep Neural Network Library (oneDNN)," Data de consulta: 23 de xuño de 2021. Dispoñible en: https://docs.oneapi.com/versions/latest/onednn/index.html

[5] P. Pröve, "An introduction to different types of convolutions in Deep Learning," *Towards Data Science*, 2017, Data de consulta: 23 de xuño de 2021. Dispoñible en: https://towardsdatascience.com/types-of-convolutions-in-deep-learning-717013397f4d

[6] A. Sharma, "Restricted Boltzmann Machines — Simplified," *Towards Data Science*, 2018, Data de consulta: 23 de xuño de 2021. Dispoñible en: https://towardsdatascience.com/restricted-boltzmann-machines-simplified-eab1e5878976

[7] T. Gupta, "Deep Learning: Feedforward Neural Network," *Towards Data Science*, 2017, Data de consulta: 23 de xuño de 2021. Dispoñible en: https://towardsdatascience.com/deep-learning-feedforward-neural-network-26a6705dbdc7

[8] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *CoRR*, vol. abs/1710.05941, 2017, Data de consulta: 23 de xuño de 2021. Dispoñible en: https://arxiv.org/abs/1710.05941

[9] D. Yu, H. Wang, P. Chen, and Z. Wei, "Mixed pooling for convolutional neural networks," in *The 9th International Conference on Rough Sets and Knowledge Technology*, 10 2014, pp. 364–375, Data de consulta: 23 de xuño de 2021. Dispoñible en: https://www.researchgate.net/publication/300020038_Mixed_Pooling_for_Convolutional_Neural_Networks

[10] 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, Data de consulta: 23 de xuño de 2021. Dispoñible en: https://ieeexplore.ieee.org/document/8721631

[11] P. San Juan, A. Castelló, M. F. Dolz, P. Alonso-Jordá, and E. S. Quintana-Ortí, "High performance and portable convolution operators for multicore processors," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020, pp. 91–98, Data de consulta: 23 de xuño de 2021. Dispoñible en: https://ieeexplore.ieee.org/document/9235053

[12] V. Strassen, "Gaussian elimination is not optimal," *Numerische Mathematik*, vol. 13, pp. 354–356, 1969, Data de consulta: 23 de xuño de 2021. Dispoñible en: https://doi.org/10.1007/BF02165411

[13] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," *Journal of Symbolic Computation*, vol. 9, no. 3, pp. 251–280, 1990, computational algebraic complexity editorial. Data de consulta: 23 de xuño de 2021. Dispoñible en: https://www.sciencedirect.com/science/article/pii/S0747717108800132

[14] J. Alman and V. V. Williams, "A refined laser method and faster matrix multiplication," *CoRR*, vol. abs/2010.05846, 2020, Data de consulta: 23 de xuño de 2021. Dispoñible en: https://arxiv.org/abs/2010.05846

[15] Shmuel Winograd, *Arithmetic complexity of computations*. Siam, 1980, vol. 33.

[16] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," *CoRR*, vol. abs/1509.09308, 2015, Data de consulta: 23 de xuño de 2021. Dispoñible en: https://arxiv.org/abs/1509.09308

[17] E. Weisstein, "Convolution Theorem," MathWorld - A Wolfram Web Resource, Data de consulta: 23 de xuño de 2021. Dispoñible en: https://mathworld.wolfram.com/ConvolutionTheorem.html

[18] Khronos, "Sycl registry," Data de consulta: 23 de xuño de 2021. Dispoñible en: https://www.khronos.org/registry/SYCL/

[19] I. O. for Standardization (ISO), "Programming languages — c++. iso/iec14882:2017," 2017, Data de consulta: 23 de xuño de 2021. Dispoñible en: https://www.iso.org/standard/68564.html

[20] NVIDIA, "CUDA Toolkit," Data de consulta: 23 de xuño de 2021. Dispoñible en: https://developer.nvidia.com/cuda-toolkit

[21] Khronos Group, "OpenCL," Data de consulta: 23 de xuño de 2021. Dispoñible en: https://www.khronos.org/opencl/

[22] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014, Data de consulta: 23 de xuño de 2021. Dispoñible en: https://arxiv.org/abs/1410.0759

[23] Intel, "oneAPI Math Kernel Library (oneMKL)," Data de consulta: 23 de xuño de 2021. Dispoñible en: https://docs.oneapi.com/versions/latest/onemkl/index.html

[24] NVIDIA, "cuBLAS API Reference," 2020, Data de consulta: 23 de xuño de 2021. Dispoñible en: https://docs.nvidia.com/cuda/cublas/index.html

[25] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Transactions on Mathematical Software*, vol. 41, no. 3, pp. 14:1–14:33, Xuño 2015, Data de consulta: 23 de xuño de 2021. Dispoñible en: https://doi.acm.org/10.1145/2764454