Algoritmos rápidos para redes neuronales convolucionales

Andrew Lavin
alavin@acm.org

Scott Gray
Sistemas Nervana
sgray@nervanasys.com

Resumen

Las redes neuronales convolucionales profundas necesitan días de cálculo de la GPU para entrenarse en grandes conjuntos de datos. La detección de peatones para coches autoconducidos requiere una latencia muy baja. El reconocimiento de imágenes en teléfonos móviles se ve limitado por la escasez de recursos de procesamiento. El éxito de las redes neuronales convolucionales en estas situaciones está limitado por la velocidad cálculo. La convolución convencional basada en la FFT es rápida para filtros grandes, pero las redes neuronales convolucionales más avanzadas utilizan filtros pequeños, de 3×3 *. Presentamos una nueva clase de algoritmos rápidos para redes neuronales convolucionales que utilizan los algoritmos de filtrado mínimo de Winograd. Los algoritmos calculan la convolución de complejidad mínima sobre mosaicos pequeños, lo que los hace rápidos con filtros pequeños y tamaños de lote pequeños. Comparamos una implementación en GPU de nuestro algoritmo con la red VGG y mostramos un rendimiento puntero con tamaños de lote de 1 a 64.*

1. Introducción

Las redes neuronales convolucionales profundas (convnets) logran resultados punteros en problemas reconocimiento de imágenes [11][7]. El entrenamiento de estas redes requiere varios días de GPU, así como importantes recursos informáticos durante la clasificación. Los conjuntos de datos y modelos más grandes mejoran la precisión, pero también aumentan el tiempo de cálculo. Por lo tanto, el progreso de las redes neuronales profundas está limitado por la velocidad de cálculo de las redes.

Del mismo modo, la aplicación de redes de convección a problemas de baja latencia, como la detección de peatones en imágenes de vídeo de coches autoconducidos, está limitada por la rapidez con la que se puede clasificar un pequeño conjunto de imágenes, posiblemente una sola.

El entrenamiento distribuido de redes de convección puede lograrse dividiendo cada lote de ejemplos entre los nodos de un clúster y acumulando actualizaciones de peso entre los nodos. Los lotes de gran tamaño afectan negativamente a la convergencia de la red, por lo que el tamaño mínimo de lote que puede calcularse de forma eficiente impone un límite superior al tamaño del clúster [8, 6].

Las arquitecturas de redes convolucionales más avanzadas para el reconocimiento de imágenes utilizan redes profundas de 3×3 capas convolucionales, es-

porque consiguen mejor precisión con menos pesos que las redes poco profundas con filtros más grandes [11, 7].

Por lo tanto, existe una gran necesidad de algoritmos convnet rápidos para lotes pequeños y filtros pequeños. Sin embargo, las bibliotecas convnet convencionales requieren lotes de gran tamaño y filtros grandes para funcionar con rapidez.

Este artículo presenta una nueva clase de algoritmos rápidos para redes neuronales convolucionales basados en los algoritmos de filtrado mínimo de los que Winograd fue pionero [13]. Los algoritmos pueden reducir la complejidad aritmética de una capa convolucional hasta un factor de 4 en comparación con la convolución directa. Casi toda la aritmética se realiza mediante multiplicidades de matrices densas de dimensiones suficientes para ser calculadas eficientemente, incluso cuando el tamaño del lote es muy pequeño. Los requisitos de memoria también son menores que los del algoritmo de convolución FFT convencional. Estos factores permiten realizar implementaciones prácticas. Nuestra implementación para las GPU NVIDIA Maxwell alcanza el rendimiento más avanzado para todos los tamaños de lote medidos, de 1 a 64, con un uso máximo de 16 MB de memoria de trabajo.

2. Trabajos relacionados

La FFT y el teorema de convolución se han utilizado para reducir la complejidad aritmética de las capas convnet, primero por Mathieu *et al.* [10], luego perfeccionado por Visalache *et al.* [12] e implementado en la biblioteca cuDNN de NVIDIA [1].

El algoritmo de Strassen para la multiplicación rápida de matrices fue utilizado por Cong y Xiao [3] para reducir el número de conversaciones en una capa convnet, reduciendo así su complejidad aritmética total. Los autores también sugirieron que otras técnicas de la teoría de la complejidad aritmética podrían ser aplicables a las convnets.

Se han intentado varios enfoques para reducir la complejidad de las redes convolucionales cuantificando o aproximando de otro modo la capa convolucional. Consideramos que estos enfoques son ortogonales y complementarios a los que explotan la estructura algebraica, por lo que los declaramos fuera del ámbito de este artículo.

3. Redes neuronales convolucionales

Una capa convnet correlaciona un banco de K filtros con C canales y tamaño $R \times S$ frente a un minilote de N imágenes con C canales y tamaño $H \times W$. Denotamos elementos filtrantes como $Gk_{c,u,v}$ y elementos de imagen como $D_{i,c,x,y}$.

El cálculo de la salida de una sola capa covnnet $Y_{i,k,x,y}$ viene dada por la fórmula

$$Yi_{,k,x,y} = \sum_{c=1}^{C} \sum_{v=1}^{S} \sum_{u=1}^{R} Di,c,x+u,y+vGk_{,c,u,v} \quad (1)$$

y podemos escribir la salida de un par imagen/filtro completo como

$$Yi_{,k} = \sum_{c=1}^{C} D_{i,c} * Gk_{,c} \quad (2)$$

donde $*$ denota correlación 2D.

4. Algoritmos rápidos

Se sabe desde al menos 1980 que el algoritmo de filtrado mínimo para calcular m salidas con un filtro FIR *de r tomas*, que llamamos $F(m, r)$, requiere

$$\mu(F(m, r)) = m + r - 1 \quad (3)$$

multiplicaciones [13, p. 39]. Además, podemos anidar algoritmos 1D mínimos $F(m, r)$ y $F(n, s)$ para formar algo 2D mínimo.
ritmos para calcular $m \times n$ salidas con un filtro $r \times s$, que llamamos $F(m \times n, r \times s)$. Estos requieren

$$\begin{aligned} \mu(F(m \times n, r \times s)) &= \mu(F(m, r))\mu(F(n, s)) \\ &= (m + r - 1)(n + s - 1) \end{aligned} \quad (4)$$

multiplicaciones [14]. Podemos seguir anidando algoritmos 1D para formar algoritmos para filtros FIR multidimensionales.

Es interesante observar que en 1D, 2D y multidimensión, el algoritmo mínimo requiere un número de multiplicaciones igual al número de entradas. En otras palabras, para calcular $F(m, r)$ debemos acceder a un intervalo de $m + r - 1$ valores de datos, y para calcular $F(m \times n, r \times s)$ debemos acceder a un mosaico de $(m + r - 1) \times (n + s - 1)$ valores de datos. Por lo tanto, el algoritmo de filtrado mínimo requiere una multiplicación por entrada.

4.1. F(2x2,3x3)

El algoritmo estándar para $F(2, 3)$ utiliza $2 \times 3 = 6$ multiplicaciones. Winograd [13, p. 43] documentó el siguiente algoritmo mínimo:

$$F(2, 3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix} \quad (5)$$

donde

$$\begin{aligned} m_1 &= (d_0 - d_2)g_0 & m_2 &= (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2} \\ m_4 &= (d_1 - d_3)g_2 & m_3 &= (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2} \end{aligned}$$

Este algoritmo sólo utiliza 4 multiplicaciones y, por tanto, es mínimo según la fórmula $\mu(F(2, 3)) = 2 + 3 - 1 = 4$. También utiliza 4 sumas que implican los datos, 3 sumas y 2 multiplicaciones por una constante que implican el filtro (la suma $g0 + g2$ se puede calcular una sola vez), y 4 sumas para re-ducir los productos al resultado final.

Los algoritmos de filtrado rápido pueden escribirse en forma matricial
como:

$$Y = A^T(Gg) \odot (B^T d) \quad (6)$$

donde $\odot$ indica multiplicación por elementos. Para $F(2, 3)$, las matrices son:

$$\begin{aligned} (B^T)^0 &= \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \\ (G)^{2} &= \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} \\ A^{(T)^1} &= \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} \\ g &= \begin{bmatrix} g_0 & g_1 & g_2 \end{bmatrix}^T \\ d &= \begin{bmatrix} d_0 & d_1 & d_2 & d_3 \end{bmatrix}^T \end{aligned} \quad (7)$$

Un algoritmo mínimo 1D $F(m, r)$ se anida consigo mismo para obtener un algoritmo mínimo 2D, $F(m \times m, r \times r)$ de este modo:

$$Y = A^T[GgG^T] \odot [B^T dB]A \quad (8)$$

donde ahora g es un filtro $r \times r$ y d es un azulejo de imagen $(m + r - 1) \times (m + r - 1)$. La técnica de anidamiento puede generarse para filtros no cuadrados y salidas, $F(m \times n, r \times s)$, anidando un algoritmo para $F(m, r)$ con un algoritmo para $F(n, s)$.

$F(2 \times 2, 3 \times 3)$ utiliza $4 \times 4 = 16$ multiplicaciones, mientras que el algoritmo estándar utiliza $2 \times 2 \times 3 \times 3 = 36$. Esto supone una reducción de la complejidad aritmética de $\frac{36}{16} = 2{,}25$. La transformación de datos utiliza 32 sumas, la transformación de filtro utiliza 28 instrucciones de coma flotante, y la transformación inversa utiliza
24 adiciones.

Los algoritmos para $F(m \times m, r \times r)$ pueden utilizarse para calcular capas convnet con $r \times r$ kernels. Cada canal de imagen es dividido en baldosas de tamaño $(m + r - 1) \times (m + r - 1)$, con $r - 1$ elementos de solapamiento entre baldosas vecinas, lo que arroja

$P = \lceil H/m \rceil \lceil W/m \rceil$ mosaicos por canal, C. A continuación, se calcula $F(m \times m, r \times r)$ para cada combinación de mosaico y filtro en cada canal, y se suman los resultados de todos los canales.

Sustituyendo $U = GgG^T$ y $V = B^T dB$, tenemos:

$$Y = A^T U \odot V A \tag{9}$$

Etiquetando las coordenadas de las baldosas como (x, y), reescribimos la fórmula de la capa de convnet (2) para una sola imagen i, un filtro k y una baldosa coordenada $(\tilde{x}, \tilde{y})$ como:

$$\begin{aligned} Y_{i,k,\tilde{x},\tilde{y}} &= \sum_{c=1}^{C} D_{i,c,\tilde{x},\tilde{y}} * g_{k,c} \\ &= \sum_{c=1}^{C} A^T U_{k,c} \odot V_{c,i,\tilde{x},\tilde{y}} A \\ &= A^T \left[\sum_{c=1}^{C} U_{k,c} \odot V_{c,i,\tilde{x},\tilde{y}} \right] A \end{aligned} \tag{10}$$

Así, podemos reducir sobre C canales en el espacio de transformación, y sólo entonces aplicar la transformada inversa A a la suma. Esto amortiza el coste de la transformada inversa sobre el número de canales.

Examinamos la suma

$$M_{k,i,\tilde{x},\tilde{y}} = \sum_{c=1}^{C} U_{k,c} \odot V_{c,i,\tilde{x},\tilde{y}} \tag{11}$$

y simplificamos la notación reduciendo las coordenadas imagen/tile (i, x, y) a una sola dimensión, b. También lamentamos cada componente de la multiplicación por elementos separado, como (ξ, ν), obteniendo:

$$M_{k,b}^{(\xi,\nu)} = \sum_{c=1}^{C} U_{k,c}^{(\xi,\nu)} V_{c,b}^{(\xi,\nu)} \tag{12}$$

Esta ecuación es sólo una multiplicación de matrices, por lo que podemos escribir:

$$(M)^{(\xi,\nu)} = U^{(\xi,\nu)} V^{(\xi,\nu)} \tag{13}$$

La multiplicación de matrices tiene implementaciones eficientes en plataformas CPU, GPU y FPGA, debido a su alta intensidad computacional. Así, hemos llegado a la implementación práctica del algoritmo rápido del Algoritmo 1.

Winograd documentó una técnica para generar el algoritmo de filtrado mínimo $F(m, r)$ para cualquier elección de m y r. La construcción utiliza el teorema chino del resto para producir un algoritmo mínimo para la convolución lineal, que es equivalente a la multiplicación polinómica, y luego transpone el algoritmo de convolución lineal para producir un algoritmo de filtrado mínimo. Se remite al lector al libro seminal de Winograd [13], o al libro de Blahut [2] para un tratamiento moderno del tema. En el material suplementario ofrecemos las derivaciones de los algoritmos específicos utilizados en este artículo.

Algoritmo 1 Cálculo de la capa Convnet con el algoritmo de filtrado mínimo de Winograd $F(m \times m, r \times r)$

$P = N \lceil H/m \rceil \lceil W/m \rceil$ es el número de mosaicos de imagen.
$\alpha = m + r - 1$ es el tamaño del mosaico de entrada. Las baldosas vecinas se solapan $r - 1$.
$d_{c,b} \in \mathbb{R}^{\alpha \times \alpha}$ es la ficha de entrada b en el canal c.
$g_{k,c} \in \mathbb{R}^{r \times r}$ es el filtro k en el canal c.
G, B^T y A^T son transformadas de filtro, de datos e inversa.

$Yk_{,b} \in \mathbb{R}^{m \times m}$ es la baldosa de salida b en el filtro k.

para $k = 0$ a K **hacer**
  para $c = 0$ a C
  hacer
    $u = Gg_{k,c}G^T \in \mathbb{R}^{\alpha \times \alpha}$
    Dispersión de u en matrices U: $U_{k,c}^{(\xi,\nu)} = u_{\xi,\nu}$
para $b = 0$ a P **hacer**
  para $c = 0$ a C
  hacer
    $v = B^T d_{c,b} B \in \mathbb{R}^{\alpha \times \alpha}$
    Dispersión de v en matrices V: $V_{c,b}^{(\xi,\nu)} = v_{\xi,\nu}$
para $\xi = 0$ a α **hacer**
  para $\nu = 0$ a α **hacer**
    $(M)^{(\xi,\nu)} = U^{(\xi,\nu)} V^{(\xi,\nu)}$
para $k = 0$ a K **hacer**
  para $b = 0$ a P
  hacer
    Recoge m de las matrices M: $m_{\xi,\nu} = M_{k,b}^{(\xi,\nu)}$
    $Y_{k,b} = A^T mA$

4.2. F(3x3,2x2)

Entrenamiento de una red mediante descenso de gradiente estocástico requiere el cálculo de los gradientes con respecto a las entradas y los pesos. Para una capa convnet, el gradiente con respecto a las entradas es una convolución del error retropropagado de la capa siguiente, de dimensión $N \times K \times H \times W$, con una versión invertida de los filtros $R \times S$ de la capa. Por lo tanto, se puede calcular utilizando el mismo algoritmo que se utiliza para la propagación hacia delante.

El gradiente con respecto a los pesos es una convolución de las entradas de las capas con los errores retropropagados, produciendo $R \times S$ salidas por filtro y canal. Por lo tanto, tenemos que calcular la convolución $F(R \times S, H \times W)$, lo que es poco práctico porque $H \times W$ es demasiado grande para nuestros algoritmos rápidos. En su lugar, descomponemos esta convolución en una suma directa de convoluciones más pequeñas, por ejemplo $F(3 \times 3, 2 \times 2)$. Aquí los 4×4 mosaicos del algoritmo superponen 2 píxeles en cada dimensión, y los 3×3 resultados se suman sobre todos los mosaicos para formar $F(3 \times 3, H \times W)$.

Las transformadas para $F(3\times 3,\ 2\times 2)$ vienen dadas por:

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & -1 & 0 & 1 \\ 1 & 1 & 1 & \end{bmatrix}, \quad G = \begin{bmatrix} 1 & & \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & & 1 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 0 & 1 & 1 & \\ 0 & 1 & -1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \tag{14}$$

Con $(3+2-1)^2 = 16$ multiplicaciones frente a la convolu- directa. $3\times 3\times 2\times 2 = 36$ multiplicaciones, consigue la misma reducción de complejidad aritmética de *36/16* = 2,25 que el algoritmo de propagación hacia delante correspondiente.

4.3. F(4x4,3x3)

Un algoritmo mínimo para $F(4,\ 3)$ tiene la forma:

$$B^T = \begin{bmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & 4 & -1 & 1 & 0 \\ 0 & 4 & -4 & 1 & 1 & 0 \\ 0 & -2 & 1 & 2 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 0 \\ 0 & 4 & 0 & -5 & 0 & 1 \end{bmatrix}$$

$$G = \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ -\frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} \\ -\frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{24} & \frac{1}{12} & \frac{1}{6} \\ \frac{1}{24} & -\frac{1}{12} & \frac{1}{6} \\ 0 & 0 & 1 \end{bmatrix} \tag{15}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 1 & 1 & 4 & 4 & 0 \\ 0 & 1 & -1 & 8 & -8 & 1 \end{bmatrix}$$

La transformación de datos utiliza 13 instrucciones de coma flotante, la transformada de filtro utiliza 8, y la transformada inversa utiliza 10.

Aplicando la fórmula de anidamiento se obtiene un algoritmo mínimo para $F(4\times 4,\ 3\times 3)$ que utiliza $6\times 6 = 36$ multiplicaciones, mientras que el algoritmo estándar utiliza $4\times 4\times 3\times 3 = 144$. Esto supone una reducción de la complejidad aritmética de 4.

La transformación de datos 2D utiliza $13(6+6) = 156$ instrucciones de coma flotante, la transformación de filtro utiliza $8(3+6) = 72$, y la transformación inversa utiliza $10(6+4) = 100$.

El número de sumas y multiplicaciones constantes requeridas por las transformaciones mínimas de Winograd aumenta cuadráticamente con el tamaño del mosaico [9, p. 211]. Por tanto, para mosaicos grandes la complejidad de las transformaciones superará cualquier ahorro en el número de multiplicaciones.

La magnitud de los elementos de la matriz de transformación también in- se reduce al aumentar el tamaño de las baldosas. Esto reduce eficazmente la precisión numérica del cálculo, de modo que para grandes baldosas, las transformaciones no pueden calcularse con precisión [13, p. 28].

Las convnets requieren sorprendentemente poca precisión numérica [4, 5]. Esto implica que podemos sacrificar algo de accuracy en el cálculo del filtrado sin afectar a la precisión de la convnet. Examinamos la posibilidad de $F(6\times 6,\ 3\times 3)$ en el material suplementario.

4.4. Transformada rápida de Fourier

La transformada rápida de Fourier (FFT) puede utilizarse para produce un algoritmo de convolución en mosaico que tiene la misma forma que el Algoritmo 1. La principal diferencia es que las matrices de trans- forma se sustituyen por la FFT y la FFT inversa. La principal diferencia es que las matrices trans-formadas se sustituyen por la FFT y la FFT inversa, y la multiplicación puntual de los componentes complejos de la FFT da lugar a la convolución cíclica. Sólo $m\times n$ componentes de la $(m+r-1)\times(n+s-1)$ convolución cíclica son válidos, el resto debe descartarse, y los mosaicos deben solaparse en $r-1$ y $s-1$ para volver a calcular las salidas descartadas.

Esta técnica se denomina solapar y guardar [2, p. 195]. similitud de solapar y guardar con nuestro enfoque facilita la comparación. Con la convolución basada en FFT, la etapa de multiplicación sigue utilizando 1 multiplicación por entrada, pero ahora los operandos son números complejos. La multiplicación directa de números complejos requiere 4 multiplicaciones reales. Gracias- totalmente, un par de trucos reducen aún más la complejidad.

La transformada de Fourier de una señal real tiene Hermitian simetría, lo que reduce casi a la mitad el número de productos únicos en cada $U\odot V$. Las implementaciones de convnet basadas en FFT han explotado esta propiedad [10, 12]. En concreto, la transformada discreta de Fourier de una matriz $\alpha\times\alpha$ de valores reales puede representarse con una matriz de $\alpha\times(\lfloor\frac{\alpha}{2}\rfloor+1)$ valores complejos. Además, $U^H V^H = (UV)^H$, por lo que los productos de los valores que faltan pueden reconstruirse simplemente tomando el conjugado complejo de los valores calculados. Así pues, la etapa de multiplicación del algoritmo FFT convnet con tamaño de mosaico $\alpha = m+r-1$ requiere $N\lceil\frac{H}{\alpha}\rceil\lceil\frac{W}{m}\rceil CK\alpha(\lfloor\frac{\alpha}{2}\rfloor+1)$ com- multiplicaciones plex, o $(\lfloor\frac{\alpha}{2}\rfloor+1)/\alpha$ complejos multiplica por entrada.

Utilizando el algoritmo estándar para multiplicar complejos números, esto equivale a $4(\lfloor\frac{\alpha}{2}\rfloor+1)/\alpha > 2$ multiplicaciones reales por entrada.

Otra técnica, que hasta donde sabemos no ha utilizado en convnets, consiste en utilizar un algoritmo rápido para multiplicar números complejos con 3 multiplicaciones reales [13]:

$$\begin{aligned}(x_0 + ix_1)(y_0 + iy_1) &= [x_0y_0 - x_1y_1,\ i(x_0y_1 + x_1y_0)]. \\ &= [u_cv_a + u_av_c,\ i(u_av_c - u_bv_b)]\end{aligned} \tag{16}$$

donde

$$\begin{aligned} u_a &= x_0 & v_a &= y_{(0)} \\ u_{(b)} &= x_0 + x_1, & v_b &= y_1 \\ u_c &= x_1 - x_0 & v_c &= y_0 + y_1 \end{aligned} \tag{17}$$

Azulejos	Winograd				FFT	
	α	β'	γ'	δ'	α'	β', γ', δ'
3	9.00	-	-	-		
4	4.00	2.00	1.75	1.50		
5	2.78	3.60	2.24	2.24		
6	**2.25**	**4.33**	**2.00**	**2.78**		
8	1.78	6.50	2.23	4.38	4.44	2.42
16					2.94	4.23
32					2.42	6.24
64					**2.20**	**8.30**
128					2.10	10.37
256					2.05	12.42

Tabla 1. Complejidad aritmética normalizada de la multiplicación (α')), la transformación de datos (β')), la transformación de filtro (γ') y la transformación inversa (δ') en función del tamaño de mosaico, tanto para la convolución basada en Winograd como para la basada en FFT. F(4x4,3x3) tiene un tamaño de mosaico de 6. La convolución directa tiene un tamaño de mosaico de 3.

Un algoritmo convnet basado en FFT puede incorporar esto modificando las transformadas FFT del filtro y los datos para dar salida a las matrices de valor real (U_a, U_b, U_c) y (V_a, V_b, V_c) en lugar de las matrices de valor complejo U y V . Esto añade 2 instrucciones de coma flotante por salida a la transformación del filtro y 1 a la transformación de datos. También aumenta el consumo de memoria de cada matriz a la mitad.

A continuación, podemos calcular $M = UV$ mediante 3 llamadas a una función estándar de multiplicación de matrices reales (*por ejemplo,* SGEMM):

$$T = U_a V_c \qquad M_0 = U_c V_a + T \qquad (18)$$

$$M_1 = -U_b V_b + T, \; M = (M_0, iM_1)$$

La acumulación de la matriz temporal T se realiza utilizando SGEMM regular con $\beta = 1$ y $C = T$, con el coste de añadir 2 instrucciones de coma flotante por salida. Podemos pensar que estas instrucciones se suman al coste de la transformación inversa. La matriz temporal T aumenta el uso de memoria a la mitad, por lo que el tamaño total del espacio de trabajo es aproximadamente el doble que el de la convolución basada en FFT con CGEMM directo.

Combinando la simetría Hermitiana con CGEMM rápido nos da una etapa de multiplicación con $3(\lfloor \alpha \rfloor + 1)/\alpha > 1.5$ multiplicaciones reales por entrada. Recordemos que la etapa de multiplicación de los algoritmos Winograd es siempre 1 multiplicación real por input. Así, incluso con CGEMM rápido, la convolución base FFT debe utilizar un tamaño de mosaico significativamente mayor para rivalizar con la complejidad aritmética de los algoritmos Winograd.

Para la propia transformada FFT, consideramos el algoritmo FFT split-radix, que es el mínimo algoritmo FFT práctico cuando N es una potencia de 2 [9, p. 150]. Suponemos que la transformada FFT 2D se construye mediante la combinación fila-columna.

y tomamos prestadas las cifras de complejidad del DSP Handbook [9, pp. 173,175] para la Tabla 1.

Azulejos	FFT con Fast CGEMM			
	α'	β'	γ'	δ'
8	3.33	3.77	4.30	4.30
16	**2.20**	**6.23**	**6.82**	**6.82**
32	1.81	8.94	9.57	9.57
64	1.65	11.72	12.36	12.36
128	1.57	14.48	15.14	15.14
256	1.54	17.22	17.88	17.88

Tabla 2. Complejidad aritmética normalizada para el filtrado FFT con CGEMM rápido Complejidad aritmética normalizada para el filtrado FFT con CGEMM rápido. El CGEMM rápido utiliza 3 multiplicaciones por complejo múltiple en lugar de 4, pero tiene una sobrecarga de transformación ligeramente mayor y utiliza más memoria.

5. Análisis de la complejidad aritmética

En nuestro modelo de convnets rápidas, la complejidad aritmética de la etapa de multiplicación es:

$$M = N \lceil H/m \rceil \lceil W/n \rceil CK(m + R - 1)(n + S - 1) \quad (19)$$

Cuando $m = n = 1$, la fórmula es igual a la complejidad aritmética de la convolución directa. Por lo tanto, la convolución directa es el algoritmo mínimo para $F(1 \times 1, R \times S)$

Aunque nuestro análisis emplea convoluciones mínimas, la capa convnet en sí no es mínima porque realiza más convoluciones de las estrictamente necesarias. Podríamos reducir el número de convoluciones empleando recursiones de Strassen como en [3], pero cada recursión reduce

las 3 dimensiones de nuestras matrices a la mitad, al tiempo que proporciona sólo una reducción de $\frac{8}{7}$ en la complejidad aritmética. La matriz no pueden calcularse eficazmente si C o K es demasiado pequeña. La convolución rápida por sí sola proporciona una reducción de la complejidad aritmética de 2,25 o más, reduciendo sólo la dimensión mayor de la matriz, P. Aún así, para capas con C, K y P grandes, puede merecer la pena realizar recursiones de Strassen además de la convolución rápida. Dejamos esta cuestión para futuras investigaciones.

Para simplificar las ecuaciones, en adelante supondremos que W/m y H/n no tienen residuos. También supondremos filtros y bloques cuadrados, $R = S$ y $m = n$.

La complejidad de la multiplicación puede reescribirse como:

$$M = (m + R - 1)^2/m^2 NHWCK$$
$$= \alpha' NHWCK \qquad (20)$$

donde $\alpha = (m + R - 1)^2$ y $\alpha' = \alpha/m^2$

Las complejidades aritméticas totales de las transformaciones de datos, filtro e inversa pueden escribirse como:

$$T(D) = \beta' NHWC$$
$$T(F) = \gamma' CK \qquad (21)$$
$$T(I) = \delta' NHWK$$

donde β, γ y δ son el número de instrucciones en coma flotante utilizadas por las transformaciones correspondientes para mosaicos individuales.

Dividiendo la complejidad de cada transformación por M se obtiene su complejidad relativa:

$$\begin{aligned} T(D)/M &= \beta/(K\alpha^2 = \beta'/K \\ T(F)/M &= \gamma/(NHW\alpha^2/m^2) \\ &= \gamma/(P\alpha^2) = \gamma'/P \\ T(I)/M &= \delta/(C\alpha^2) = \delta'/C \end{aligned} \quad (22)$$

Llamamos β'), γ') y δ') a los complejos aritméticos normalizados de las transformaciones de datos, filtro e inversa, respectivamente. $P = NHW/m^2$ es el número de mosaicos por canal.

Sumando los términos de cada etapa se obtiene la complejidad aritmética total de la capa convnet:

$$L = \alpha'(1 + \beta'/K + \gamma'/P + \delta'/C)NHWCK \quad (23)$$

Para lograr una gran aceleración, la complejidad de la multiplicación α' debe ser lo más pequeña , y las complejidades de la trans- forma β', γ' y δ') deben ser pequeñas en comparación con K, P y C, respectivamente.

Para la convolución directa, $\alpha' = \alpha^2 = R^2$ y $\beta' = \gamma' = \delta' = 0$. Por tanto, el speedup máximo de un algoritmo rápido frente a la convolución directa es R^2/α'.

En las tablas 1 y 2 se muestra la complejidad normalizada de la transformación para distintos tamaños de mosaico y algoritmos. Debido a su similitud con nuestro enfoque, la complejidad de la convolución basada en FFT también puede medirse con la ecuación 23.

Las capas convnet basadas en FFT con CGEMM directo deben utilizar un tamaño de mosaico de al menos 64× 64 para igualar la complejidad de la etapa de multiplicación de Winograd F (4× 4, 3× 3) y su mosaico de 6× 6, pero entonces incurre en una sobrecarga de transformación mucho mayor. Además, un mosaico de 64× 64 desperdiciará cálculo en muchos píxeles no deseados para imágenes con tamaños que no se acercan a un múltiplo de 62× 62. Incluso para capas de tamaño moderado, se debe utilizar un minilote de moderado a grande, o habrá muy pocas baldosas para computar el CGEMM eficientemente. Por último, la memoria utilizada por un único canal de filtro transformado es de 64× 64 = 4096 unidades, lo que supone una gran expansión del filtro de 3× 3= 9 unidades. El mosaico 6x6 de F (4× 4) expande el mismo filtro a 6 6× = 36 unidades.

Las capas convnet basadas en FFT con CGEMM rápido pueden ser mucho más competitivas que los algoritmos Winograd. Tienen paridad de etapa de multiplicación con un tamaño de mosaico de 16, y una complejidad de transformación rea- sonable. Además, el tamaño de mosaico 16 genera un número razonablemente grande de mosaicos con capas convnet grandes o un tamaño de lote moderado.

Incluso con el rápido CGEMM, el mayor tamaño de los mosaicos en comparación con Winograd significa que las implementaciones convnet basadas en FFT deben tener un gran espacio de trabajo en memoria para contener los datos transformados. Para amortizar el coste de la transformación y generar matrices con un valor de dimensiones lo suficientemente grandes como para que la etapa de multiplicación sea efi- ciente. Esto resulta problemático para las GPU actuales, que disponen de una cantidad limitada de memoria en el chip. Las CPU disponen de grandes cachés y, por tanto, podrían calcular la convolución basada en FFT de forma más eficiente.

6. Implementación de la GPU

Hemos implementado F (2× 2, 3× 3) para las GPU NVIDIA Maxwell y lo hemos probado en el modelo NVIDIA Titan X.

El pequeño tamaño de las baldosas de 4× 4 y las transformaciones ligeras de F (2× 2, 3× 3) hacen posible una implementación fusionada de las etapas del algoritmo, en la que la transformación de los datos y del filtro, las 16 multiplicaciones de matrices por lotes (GEMM) y la trans- forma inversa se calculan en el mismo bloque. Otra limitación de recursos es la caché de instrucciones, en la que sólo caben unas 720 instrucciones. Nuestro bucle principal mayor, pero alinear el inicio del bucle con el límite de 128 bytes de la línea de caché de instrucciones ayuda a mitigar el coste de una pérdida de caché.

Los 16 GEMM por lotes computan 32× 32 salidas, lo que nos permite encajar el espacio de trabajo en los registros y la memoria compartida de un único bloque y seguir teniendo 2 bloques activos por SM para ocultar la latencia. El relleno cero está implícito en el uso de predicados. Si el predicado deselecciona una carga de imagen global, el valor cero se carga con una instrucción I2I de doble emisión.

Los datos de imagen se almacenan en orden CHWN para facilitar las cargas de memoria con- tiguas y alineadas, reduciendo significativamente el over-fetch. Empleamos una estrategia de "superbloqueo" para cargar 32 mosaicos de tamaño 4× 4 a partir de un número configurable de imágenes, filas y columnas. Para N>= 32, cargamos mosaicos de 32 imágenes separadas. Para N< 32, cargamos un superbloque de X× $Y = 32/N$ mosaicos por imagen. Esta estrategia facilita cargas eficientes con tamaños de lote pequeños, ya que las dimensiones W× N de los datos de entrada son contiguas en la memoria. Además, el solapamiento de 2 píxeles entre mosaicos adyacentes provoca altas tasas de aciertos en la caché L1 cuando se utilizan varios mosaicos en un .

También empleamos el bloqueo de caché L2 para aumentar la reutilización de bloques solapados. Dado que el número de mosaicos de imagen suele ser mucho mayor que el número de filtros, nuestro mapeo de bloques itera sobre un grupo de hasta 128 filtros en el bucle interno y, a continuación, itera sobre todos los mosaicos de imagen en el segundo bucle. Todos los canales del grupo de filtros caben en la caché L2, por lo que cada filtro sólo se cargará una vez desde la memoria DDR, y cada mosaico de imagen se cargará ⌈K/128⌉ veces mientras iteramos sobre los grupos de filtros. Esta estrategia reduce el ancho de banda de la memoria DDR casi a la mitad.

Hemos implementado una versión de nuestro núcleo que carga datos fp16, lo que disminuye el ancho de banda de la memoria global. También implementamos una variante que llamamos "FX" que ejecuta primero un núcleo de transformación de filtro y almacena el resultado en un búfer del espacio de trabajo. El núcleo de convolución carga los valores del filtro transformado desde el espacio de trabajo según sea necesario. El tamaño del espacio de trabajo es de sólo $16KC$ unidades de memoria, lo que equivale a sólo 16MB cuando K C= = 512 y los datos son fp32.

Capa	Profundidad	$C\times H\times W$	K	GFLOPs
conv 1.1	1	3××224	64	0.17
conv 1.2	1	64× 224× 224	64	3.70
conv 2.1	1	64× 112× 112	128	1.85
conv 2.2	1	128× 112× 112	128	3.70
conv 3.1	1	128× 56× 56	256	1.85
conv 3.2	3	256× 56× 56	256	11.10
conv 4.1	1	256× 28× 28	512	1.85
conv 4.2	3	512× 28× 28	512	11.10
conv 5	4	512× 14× 14	512	3.70
Total				39.02

Tabla 3. Capas de convolución de la red VGG E. Todas las capas utilizan 3 filtros× 3. La profundidad indica el número de veces que una determinada forma de capa aparece en la red. Los GFLOPs se ponderan por la profundidad y se asume que N=1.

7. Experimentos

Realizamos experimentos de precisión y velocidad con VGG Network E [11]. Se trata de una red profunda que utiliza 3× 3 filtros exclusivamente en las capas de convolución, que se resumen en la Tabla 3.

Probamos la precisión de nuestros algoritmos rápidos tanto con datos y filtros de precisión simple (fp32) como de precisión media (fp16). En todas las pruebas utilizamos instrucciones aritméticas fp32. Utilizamos datos y filtros aleatorios de la distribución uniforme [− *1,* 1] y medimos el error de elemento absoluto. La verdad sobre el terreno se calculó mediante convolución directa utilizando un acumulador de doble precisión para las reducciones.

Hemos medido la velocidad de nuestra implementación de *F* en la GPU (2× 2, 3× 3) y la hemos comparado con cuDNN v3 [1] en una GPU NVIDIA Titan X su- perclockeada. Desactivamos el boost clock y observamos una velocidad de reloj máxima de 1126 MHz. La GPU tiene 3072 núcleos, lo que supone un rendimiento máximo del dispositivo de 2× 3072× 1126= *6,*96 TFLOPS.

La velocidad de una capa determinada se calculó dividiendo el número de GFLOPs de cálculo requeridos por la con- volución directa, como se tabula en 3, por el tiempo de ejecución en milisegundos para obtener los TFLOPS efectivos. La reducción de complejidad aritmética permite que los algoritmos rápidos tengan TFLOPS efectivos que pueden superar el rendimiento máximo del dispositivo.

El total de GFLOPs y el tiempo de ejecución se calcularon ponderando los GFLOPs y el tiempo de ejecución de cada capa por su profundidad, y el rendimiento total se calculó como el cociente de ambos.

8. Resultados

La Tabla 4 muestra la precisión numérica de los distintos algoritmos de capa de con- volución probados con datos de entrada y filtros de precisión simple (fp32) y media precisión (fp16).

F (2× 2, 3× 3) es en realidad algo más preciso que la convolución directa. Sus transformaciones simples no pierden mucha precisión, y su etapa de multiplicación realiza una reducción sobre los canales *C*, en lugar de los elementos de filtro *RSC* obtenidos por convolución directa. *F* (4× 4, 3× 3) tiene un error mayor, pero sigue siendo más preciso que la convolución directa con datos fp16.

Todos los algoritmos probados son igual de precisos con datos fp16. Aquí la precisión está limitada por la precisión de las entradas. Dado que la convolución directa es suficientemente precisa para el entrenamiento y la inferencia con datos de baja precisión [4, 5], concluimos que *F* (4× 4, 3× 3) también lo es.

La Tabla 5 y la Tabla 6 muestran el rendimiento total de las capas E de la red VGG para cuDNN y nuestra implementación *F* (2× 2, 3× 3) para datos fp32 y fp16 para diferentes tamaños de lote.

Para los datos fp32, *F* (2× 2, 3× 3) es 1,48X en *N*= 64 y 2,26 veces más rápido con *N*= 1. El rendimiento con *N*= 16 es de 9,49 TFLOPS. Para datos fp16, F (2× 2, 3× 3) amplía su ventaja sobre cuDNN, regis*t*rando un rendim*ie*nto de 10,28 TFLOPS para N = 64. El rendimiento para N 8 sigue siendo muy bueno, con 9,57 TFLOPS. El rendimiento de *N*= 8 sigue siendo muy bueno con 9,57 TFLOPS.

La figura 1 muestra el rendimiento por capa. Las marcas de sombreado indican las capas en las que cuDNN utilizó el algoritmo FFT, de lo contrario se utilizó la convolución directa. Para *F* (2× 2, 3× 3), las marcas indican que se utilizó la transformada de filtro externo (FX), de lo contrario la transformada fusionada fue más rápida.

cuDNN seleccionar erróneamente su algoritmo FFT para valores intermedios de *N* a pesar de que su rendimiento es muy pobre, por debajo de 2 TFLOPS. Aunque probablemente se trate de un error, es revelador. El bajo rendimiento en valores moderados de *N* sugiere que la implementación de la convolución FFT utiliza mosaicos grandes, o posiblemente un único mosaico por imagen, como en [12], lo que conduce a etapas de multiplicación ineficientes a menos que *N* sea grande. Con *N* grande, la FFT cuDNN rinde mucho mejor, pero se mantiene por debajo de los 8 TFLOPS.

F (2× 2, 3× 3) funciona mejor que cuDNN en todas las capas y tamaños de lote, excepto en la capa conv1.1, que contribuye con menos del 0,5% del cálculo total de la red.

En general, comprobamos que la variante FX de nuestra aplicación funcionaba mejor a menos que el número de filtros y canales fuera muy grande. El cálculo de la transformación del filtro está muy limitado por la memoria, por lo que transformar un banco de filtros más grande disminuye la eficiencia computacional.

El peor rendimiento *F* (2× 2, 3× 3) se produce para las 14× 14 capas cuando *N*= 1. En este caso el superbloque 8× 4 se ejecuta sobre el límite de la imagen y computa píxeles no deseados. El rendimiento en esta configuración de capas sigue siendo superior a 5 TFLOPS, mientras que el rendimiento de cuDNN es de sólo 1,6 TFLOPS. cuDNN FFT utiliza un espacio de trabajo de memoria global de hasta 2,6 GB en nuestros experimentos. Por el contrario, nuestra implementación de *F* fusionada (2× 2, 3× 3) no utiliza ningún espacio de trabajo global, y la variante FX no utiliza más de 16 MB.

El rendimiento de *F* (2× 2, 3× 3) muestra nuevas capacidades de alto rendimiento y pequeño tamaño de lote con redes neuronales convolucionales de última generación. Esperamos que el rendimiento vuelva a aumentar cuando se implemente *F* (4× 4, 3× 3).

Capa	fp32			fp16
	Directo	F(2x2,3x3)	F(4x4,3x3)	
1.2	4.01E-05	1.53E-05	2.84E-04	1.14E-02
2.2	8.01E-05	2.86E-05	5.41E-04	1.45E-02
3.2	1.53E-04	5.34E-05	9.06E-04	1.99E-02
4.2	3.20E-04	5.34E-05	1.04E-03	3.17E-02
5	3.43E-04	4.20E-05	1.08E-03	2.61E-02

Tabla 4. Error máximo de elemento en las capas de la red VGG. Con datos fp32, $F(2\times 2, 3\times 3)$ es más preciso que la convolución directa. Con datos fp16, todos los algoritmos son igual de precisos.

N	cuDNN		F(2x2,3x3)		Aceleración
	mseg	TFLOPS	mseg	TFLOPS	
1	12.52	3.12	5.55	7.03	2.26X
2	20.36	3.83	9.89	7.89	2.06X
4	104.70	1.49	17.72	8.81	5.91X
8	241.21	1.29	33.11	9.43	7.28X
16	203.09	3.07	65.79	9.49	3.09X
32	237.05	5.27	132.36	9.43	1.79X
64	394.05	6.34	266.48	9.37	1.48X

Tabla 5. Rendimiento de cuDNN frente a $F(2\times 2, 3\times 3)$ en la red VGG E con datos fp32. El rendimiento se mide en TFLOPS efectivos, la relación entre los GFLOPs directos del algoritmo y el tiempo de ejecución.

N	cuDNN		F(2x2,3x3)		Aceleración
	mseg	TFLOPS	mseg	TFLOPS	
1	14.58	2.68	5.53	7.06	2.64X
2	20.94	3.73	9.83	7.94	2.13X
4	104.19	1.50	17.50	8.92	5.95X
8	241.87	1.29	32.61	9.57	7.42X
16	204.01	3.06	62.93	9.92	3.24X
32	236.13	5.29	123.12	10.14	1.92X
64	395.93	6.31	242.98	10.28	1.63X

Tabla 6. Rendimiento de cuDNN frente a $F(2\times 2, 3\times 3)$ en la red VGG E con datos fp16.

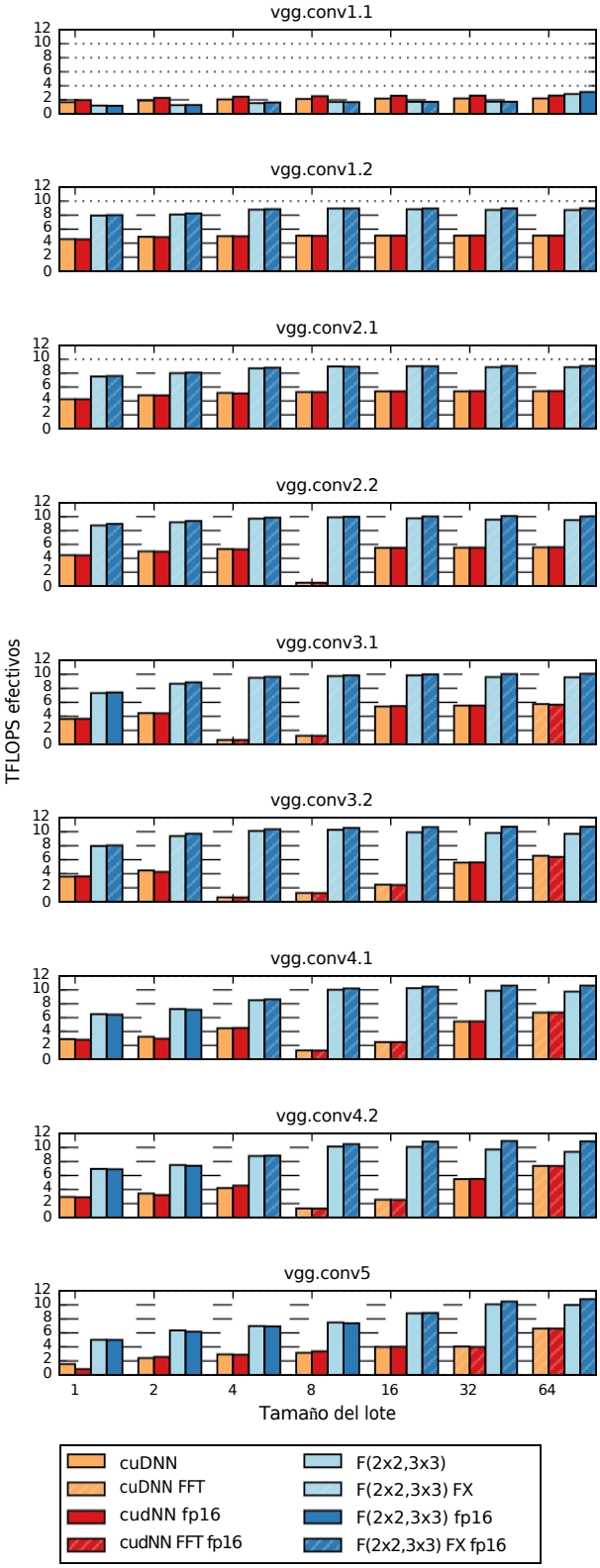

Figura 1. TFLOPS efectivos netos de VGG frente al tamaño del lote para cuDNN y $F(2\times 2, 3\times 3)$ en una GPU NVIDIA Titan X de 6,96 TFLOPS.

Referencias

[1] cuDNN. https://developer.nvidia.com/cudnn. Consultado: 2015-11-01. 1, 7

[2] Richard E. Blahut. *Fast algorithms for signal process-* ing. Cambridge University Press, 2010. 3, 4

[3] Jason Cong y Bingjun Xiao. Minimizing compu- tation in convolutional neural networks. En *Artifi- cial Neural Networks and Machine Learning-ICANN 2014*, páginas 281-290. Springer, 2014. 1, 5

[4] Matthieu Courbariaux, Yoshua Bengio y Jean-Pierre David. Low precision arithmetic for deep learn- ing. *CoRR*, abs/1412.7024, 2014. 4, 7

[5] Suyog Gupta, Ankur Agrawal, Kailash Gopalakr- ishnan y Pritish Narayanan. Deep learning with limited numerical precision. *arXiv preprint arXiv:1502.02551*, 2015. 4, 7

[6] Suyog Gupta, Wei Zhang y Josh Milthrope. Model accuracy and runtime tradeoff in distributed deep learning. *arXiv preprint arXiv:1509.04210*, 2015. 1

[7] Sergey Ioffe y Christian Szegedy. Batch nor- malization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015. 1

[8] Alex Krizhevsky. One weird trick for paralleliz- ing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014. 1

[9] V. Madisetti. *The Digital Signal Processing Hand-book*. Número v. 2 en Electrical engineering handbook series. CRC, 2010. 4, 5

[10] Michae¨l Mathieu, Mikael Henaff y Yann LeCun. Entrenamiento rápido de redes convolucionales mediante ffts. *CDR*, abs/1312.5851, 2013. 1, 4

[11] Karen Simonyan y Andrew Zisserman. Very deep convolutional networks for large-scale image recogni- tion. *arXiv preprint arXiv:1409.1556*, 2014. 1, 7

[12] Nicolas Vasilache, Jeff Johnson, Michae¨l Mathieu, Soumith Chintala, Serkan Piantino y Yann LeCun. Redes convolucionales rápidas con fbfft: A GPU perfor- mance evaluation. *CoRR*, abs/1412.7580, 2014. 1, 4, 7

[13] Shmuel Winograd. *Arithmetic complexity of computa- tions*, volumen 33. Siam, 1980. Siam, 1980. 1, 2, 3, 4

[14] Shmuel Winograd. On multiplication of polynomials modulo a polynomial. *SIAM Journal on Computing*, 9(2):225-229, 1980. 2