An Efficient Polynomial Multiplication Derived Implementation of Convolution in Neural Networks

Haoke Xu
University of Delaware
Newark, USA
xhaoke@udel.edu

Yulin Zhang
Minzu University of China
Beijing, China
yzhang@muc.edu.cn

Zitong Cheng
University of Delaware
Newark, USA
ztcheng@udel.edu

Xiaoming Li
University of Delaware
Newark, USA
xli@udel.edu

Abstract

Convolution is the most time consuming computation kernel in Convolutional Neural Network (CNN) applications and the majority of Graph Neural Network (GNN) applications. To achieve good convolution performance, current NN libraries such as cuDNN usually transform the naive convolution problem into either the matrix vector multiplication form, called the *im2col+MM* approach, or into the Fourier domain representation, namely the FFT approach. Both approaches either introduce significant amount of data redundancy to reduce the number of data passes and to leverage highly tuned linear algebra libraries, or avoid data redundancy but require multiple passes on input. Therefore, no implementation of convolution can outperform others in all cases. In this paper, we introduce a novel transformation that reinterprete the convolution in NN as a polynomial multiplication problem. By carefully constructing a number of conceptual polynomials, NN's convolution essentially becomes a well-known problem of calculating the coefficients in the product of two polynomials. We evaluate our approach in multiple ways: at the API level comparing it with one of the most widely used NN libraries cuDNN, as well as implementing it in PyTorch and comparing the performance with benchmark networks. On three NVIDIA GPUs 3090Ti, V100 and A10G, our approach outperforms cuDNN over a broad range of parameters with the max speedups of 34.6%, 43.1% and 33.6% on these GPUs, respectively.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CGO '25, March 01–05, 2025, Las Vegas, NV, USA
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1275-3/25/03
https://doi.org/10.1145/3696443.3708947

CCS Concepts: • **Software and its engineering** → **Software performance**; • **Theory of computation** → **Mathematical optimization**.

Keywords: Convolution, Machine Learning, Polynomial Multiplication, FFT

ACM Reference Format:
Haoke Xu, Yulin Zhang, Zitong Cheng, and Xiaoming Li. 2025. An Efficient Polynomial Multiplication Derived Implementation of Convolution in Neural Networks. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3696443.3708947

1 Introduction

It is well known that in many neural network types, the most time consuming computation step is the convolution. For example, Convolutional Neural Network [11], a widely used network type, typically spend over 92% [5, 7, 8, 13, 16, 19, 20] of the total execution time on convolution. As another example, among Graph Neural Network (GNN) applications [1], an increasingly popular form of neural network, the majority have convolution as the performance bottleneck [2].

Therefore, optimizing convolution's performance naturally becomes a focus in the efforts that try to improve the implementation efficiency of neural networks (NN). Here we briefly overview how convolution is implemented in neural network libraries, and discuss the data/memory efficiency and operational efficiency of the typical technical paths. This discussion will provide a context for us to introduce our work, and help explaining where we innovate and improve.

We would like to focus the comparison on two important performance parameters: data redundancy and operational efficiency. Here the data redundancy means the duplication of data in an approach. In other words, how a value, either from the network input or from the filter kernels, is represented and/or stored multiple times in an implementation. Data redundancy is typically introduced to facilitate the translation of convolution into other computation routines. Even the redundant data may be only conceptually represented and not be actually created in memory, the number of memory transfers required is still determined by the conceptual redundancy. On today's computer systems, the transfer of data can be more expensive than the storage of it. Furthermore, the operational efficiency can be divided into the degree of operation redundancy and practical efficiency. The operation redundancy means how the same information is computed wholly or partially multiple times. The practical efficiency means whether the operations, and the derived memory access patterns, can be efficiently implement on today's computer systems.

First as the baseline, let's look at the the naive definition of convolution in 2D. With input I of dimension $[I_h, I_w]$ and filter K of dimension $[K_h, K_w]$, convolution is defined as $conv_{2D}(I,K)_{(i_h,i_w)} = \sum_{k_h=0}^{K_h-1}\sum_{k_w=0}^{K_w-1} I[i_h+k_h, i_w+k_w] * K[k_h,k_w]$. Practical implementations of convolution, however, do not follow this naive definition because it implies memory access patterns that are highly inefficient on today's computer architectures.

In the field of neural networks, the practical implementations of 2D convolution are generally based on two methods- $im2col + MM$ [3, 8] and $FFT - based$ [14]. For example, cuDNN, one of the widest used NN libraries, employs both methods and their variants such as the Winograd method [10] to implement 2D convolution. The two methods and their variants are not universally optimal, i.e., in some scenarios, one is better than others and vice versa. This is because the methods have different degrees of data redundancy and operational efficiency that are determined by parameters such as input size, kernel size, etc.

The $im2col+MM$ method tries to take advantage of highly tuned matrix multiply (MM) libraries to implement convolution. To transform the problem, the *im2col* process unrolls and duplicates the input matrix elements in a way that convolution can be done by multiplying the transformed matrix with the unrolled kernel vector. The *im2col* method benefits from the high-performance MM libraries such as cuBLAS, but pays the hefty price of high data redundancy. The FFT-based method uses the well-known theory that convolution can be converted to element-wise multiplication in the Fourier domain. Therefore, FFT-based method reduces the time complexity of 2D convolutions but requires performing 1D FFT, and also the inverse FFT, on each row and column of the matrices. It has complex computation flows. Especially for smaller input and kernel sizes, the FFT-based methods are generally less efficient than the *im2col* method.

A recently developed approach [21] takes advantage of the discovery that the *im2col* transformed input forms a doubly blocked Hankel [17] matrix, therefore the multiplication of such a matrix with kernels can be improved by converting it to iterative finer-grain block level FFTs. However, this approach still conducts redundant FFTs on the block level, and requires data padding for each block to the next power-of-two size. Essentially a portion of data redundancy is transformed into some computational redundancy. If well tuned, it can achieve a better tradeoff.

To summarize the existing implementation paths, they either introduce data redundancy to convert convolution into a simpler and more efficiently computation routine, or avoid data duplication but resolve to multiple passes of data and information-wise redundant operation flows.

Our starting point was a curious observation of a missing piece in the state of the art in convolution implementation. It is widely known that beside the linear algebra interpretation, i.e., the *im2col* method, and the Fourier domain interpretation, i.e., the *FFT*-based method, convolution can also be interpreted as a polynomial multiplication coefficient-finding problem. That is, the input data and the kernel in convolution can be mapped into two polynomials, and the convolution between the input and the kernel will become the coefficients in the multiplication of the input polynomial and the kernel polynomial.

However, while theoretically equivalent to *im2col* and *FFT* methods, no practical convolution implementations seem to adopt this polynomial interpretation. For example, *cuDNN* has eight different methods that users can choose from. Among the methods , four are *im2col* and its variants, and the others are *FFT* and the variants.

The missing of the polynomial interpretation is not from negligence, we believe. The reason is that it is hard to come up with an efficient mapping from convolution's input and kernel to the polynomials. First of all, while the 1D convolution has a simple and naive mapping into the polynomial domain, the construction of a valid mapping for 2D is not arbitrary nor trivial. Secondly, a valid mapping essentially determines how memory is accessed, as well as the amount and sequence of operation. This makes it more challenging to find a valid mapping that is also efficient to implement. For example, the simplest, generic polynomial interpretation of 2D convolution essentially would be equivalent to a 2D FFT problem. As such, its implementation would be same as the existing *FFT*-based implementation of convolution. In other words, while the polynomial interpretation provides a new way to represent convolution, the generic representations/mappings do not provide any advantage in performance. This is probably why such a powerful conceptual model of convolution is not leveraged by practitioners.

We realized that the potential of polynomial representation hasn't really been touched. Our insight is that the generic polynomial mapping of 2D convolution fails to leverage the intrinsic structures in the handling of input and kernel in convolution. A mapping tailored for these intrinsic structures, if can be correctly constructed, will naturally lead to higher implementation efficiency for convolution.

Our key idea, as well as **our main contribution**, is that we meticulously construct a conceptual polynomial for the *im2col* matrix and another polynomial for the kernel in

Table 1. Summary of Notations

Name	Description
K	Number of kernels
I_h	Input height
I_w	Input width
K_h	Kernel height
K_w	Kernel width
O_h	Output height
O_w	Output width
P	Padding
C	Input channels
N	Mini-batch size

a special way, so that convolution, originally in the form of the multiplication between the *im2col* matrix and the kernel, can be converted into the form of an implementation-wise efficient polynomial multiplication coefficient-finding problem. Importantly, the polynomial coefficient problem in our work can then be solved as a *single* $1D$ FFT problem. Therefore, we can achieve better performance by avoiding the data redundancy in the *im2col* method and the iterative operation redundancy in the $2D - FFT$-based method.

More specifically, our work is based on two known signal/data processing and algorithm design techniques, i.e., the polynomial interpretation of linear algebra [6, 12], and the Fourier domain solution for polynomial coefficient problem. We tailor the first technique specifically for the *im2col* matrix, and use the second technique to build the actual FFT based implementation. These polynomials are carefully designed so that the coefficients of certain terms in their product are precisely the results of multiplying the *im2col* matrix with the unfolded kernel, which also equals correspondingly to the 2D convolution results. Notably, the *im2col* form of input and the polynomials are only needed in the conceptual deduction, and are not actually built in implementation. Therefore, our method doesn't need to redundantly expand input and kernel, nor need to compute FFT and inverse FFT for each row and column separately. Essentially, our "secret sauce" is that in total we only need to conduct one FFT on the original (not expanded) input, one element-wise multiplication, and one inverse FFT to calculate convolution. As the result, we minimize the number of passes over the input, and require much less memory storage or transfer overhead than what is needed in the *im2col*, FFT or the Hankel matrix based methods.

2 Method

Let's first briefly introduce the overall working flow of how we calculate convolution. To help relate to other work, the deep- learning domain notations used in this paper are summarized in Table 1.

Our work starts with a conceptual transformation of the convolution into the multiplication between a matrix and a vector. The matrix is transformed from the input and the vector transformed from the kernel in neural network. As motioned earlier, this matrix-vector based approach is referred as the *im2col* + *MM* method and is not new here. Although our idea is based on the *im2col* transformation, we want to highlight that *im2col* serves only as a purely conceptual construction that our work derives from. In our implementation, we actually never conduct the *im2col* step to the input or the filter, so not suffering from the redundancy associated with *im2col*.

The second step is the transformation of the matrix-vector multiplication form of convolution into a polynomial multiplication problem. This is the key contribution of our work in this paper. Zhang et.al report that the matrices generated from *im2col* all have doubly layered redundancy patterns, and both layers follow the definition of the Hankel matrix [17]. We realize that the doubly Hankel pattern of the matrix make it possible to construct an interesting mapping from the element indices in the matrix to the exponents in a virtual polynomial, as well as a mapping for the kernel-derived vector, so that the matrix-vector multiplication form of convolution is transformed into the multiplication of two polynomials. And very interestingly, the mappings can be constructed in such a way that the coefficients of the result polynomial are exactly the result of the convolution that is intended to be calculated in the first place.

The third step of our work is how we solve the polynomial multiplication form of convolution. The generic polynomial multiplication is a well known problem, and its solution using FFT is also a well known technique. In our work, we mainly follow the known FFT solution with some implementation details adapted for the convolution-derived specifics.

2.1 Redundancy of im2col Matrix

In this section, we will explain the im2col process and why the im2col matrix constructed is a doubly blocked Hankel matrix. This information is crucial as it provides the necessary background for the introduction of how we conceptually map the *im2col* result matrix into polynomials.

The im2col operation stands for "image to column". Its goal is to reformat the input matrix in NN, so that the convolution in NN can be converted into matrix-vector multiplication. It stretches or unrolls the local receptive field or patch that the kernel slides over into columns. Each column corresponds to a different patch of the input data. As the kernel slides in 2D convolution, the im2col operation is applied at each position of the kernel overlayed on the input. The generated columns are placed side by side and concatenated with other columns to form a matrix. The number of columns in this matrix is equal to the number of positions the kernel can take as it slides over the input. Thus, an input image is transformed into a matrix where each column represents a patch and the convolution can then be performed as matrix multiplication.

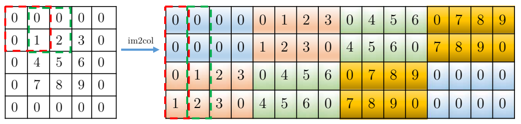


Figure 1. A 3×3 input with zero-padding of size 1 convolves with a 2 × 2 kernel (shown in dotted lines on the left). On the right is the unrolled matrix generated by the im2col process.

The matrices that the im2col operation generates have intricate patterns, and also introduce degree of data redundancy. Due to the overlap in the sliding window operation, the unrolled columns have many elements in common. Specifically, as each row of the kernel slides horizontally, the adjacent unrolled columns differ only in one element corresponding to the new element added at the right side of the row kernel. It creates a pattern that resembles a Hankel matrix. Hankel matrix is a type of matrix in which each value along ascending skew-diagonals are constant. In im2col, the kernel moves one step at a time across the input horizontally until it reaches the rightmost position, then the kernel resets to the leftmost position on the subsequent row below, and the kernel continues moving from left to right. Similarly, the corresponding columns repeats most of their elements differing only in the elements corresponding to the bottom row of the kernel. In this repetitive pattern, a doubly blocked Hankel matrix is formed. It is doubly blocked because not only the overall structure consisting of the blocks formed by the kernel sweeping possesses the Hankel pattern, but also each individual block is a Hankel matrix.

Figure 1 illustrates the convolution of a 3 × 3 input matrix with a zero-padding of size 1 using a 2 × 2 kernel. Initially the kernel is positioned at the top-left corner of the input (indicated by the red dotted line), the kernel slides one step to the right (green dotted line). The corresponding unrolled columns are represented on the right side of the figure by matching dotted lines. As the kernel continues to slide to the bottom right corner of the input, the full unrolled matrix is formed. Clearly, this figure reveals the intrinsic data pattern as a doubly Hankel matrix. Each block that shares the same color is identical, and elements within each block remain constant along the skew diagonals.

Hankel matrices have many nice properties that may be used to simplify calculation or reduce memory storage. The most useful property for our work is that Hankel matrices are structured and can be described more concisely than the n^2 elements in a $n * n$ matrix, without any loss of information. Next we will use this concise representation of the doubly blocked Hankel matrices to design equally concise polynomial representation of these specific matrices, so that the conceptual multiplication of the matrices with kernel vectors can be reformed into the multiplication of the concise polynomial representation of the matrices and the constructed polynomials for the kernel vectors.

2.2 Polynomial Construction

Next we will describe how we construct the polynomials multiplication problem for convolution. We will start with going through our approach with a small convolution problem, and then develop the example into a general solution.

Let us start with a simplest neural network problem involving a 5 × 5 image and a 3 × 3 kernel. The input image can be represented as:

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix}$$

and the kernel as:

$$U = \begin{pmatrix} u_{0,0} & u_{0,1} & u_{0,2} \\ u_{1,0} & u_{1,1} & u_{1,2} \\ u_{2,0} & u_{2,1} & u_{2,2} \end{pmatrix}$$

The convolution between A and U, if there is no padding, then will produce a 3 × 3 matrix D, as:

$$O_{w/h} = I_{w/h} - K_{w/h} + 1$$

Following the naive definition of convolution, we have $D_{i,j} = conv_{2D}(A, U)_{(i,j)} = \sum_{u=0}^{U-1} \sum_{v=0}^{V-1} A[i+u, j+v] * U[u, v]$.

The naive definition of D is not good for actual implementation, because it requires multiple passes over the input and the kernel matrices and implies memory accesses with varying strides, both of which will drag on performance. Convolution can be transformed into a matrix-vector multiplication after we expand and duplicate the input matrix through the *im2col* process. In our example, the transformed input matrix A_{im2col} will look like the following:

$$A_{im2col} = \begin{pmatrix} \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{0,1} & a_{0,2} & a_{0,3} \\ a_{0,2} & a_{0,3} & a_{0,4} \end{pmatrix} \begin{pmatrix} a_{1,0} & a_{1,1} & a_{1,2} \\ a_{1,1} & a_{1,2} & a_{1,3} \\ a_{1,2} & a_{1,3} & a_{1,4} \end{pmatrix} \begin{pmatrix} a_{2,0} & a_{2,1} & a_{2,2} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{2,2} & a_{2,3} & a_{2,4} \end{pmatrix} \\ \begin{pmatrix} a_{1,0} & a_{1,1} & a_{1,2} \\ a_{1,1} & a_{1,2} & a_{1,3} \\ a_{1,2} & a_{1,3} & a_{1,4} \end{pmatrix} \begin{pmatrix} a_{2,0} & a_{2,1} & a_{2,2} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{2,2} & a_{2,3} & a_{2,4} \end{pmatrix} \begin{pmatrix} a_{3,0} & a_{3,1} & a_{3,2} \\ a_{3,1} & a_{3,2} & a_{3,3} \\ a_{3,2} & a_{3,3} & a_{3,4} \end{pmatrix} \\ \begin{pmatrix} a_{2,0} & a_{2,1} & a_{2,2} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{2,2} & a_{2,3} & a_{2,4} \end{pmatrix} \begin{pmatrix} a_{3,0} & a_{3,1} & a_{3,2} \\ a_{3,1} & a_{3,2} & a_{3,3} \\ a_{3,2} & a_{3,3} & a_{3,4} \end{pmatrix} \begin{pmatrix} a_{4,0} & a_{4,1} & a_{4,2} \\ a_{4,1} & a_{4,2} & a_{4,3} \\ a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix} \end{pmatrix} \tag{1}$$

Correspondingly, we need to fully flatten the kernel matrix into a vector as defined in Eq. 2.

$$U_{im2col} = \begin{pmatrix} u_{0,0} & u_{0,1} & u_{0,2} & u_{1,0} & u_{1,1} & u_{1,2} & u_{2,0} & u_{2,1} & u_{2,2} \end{pmatrix}^T \tag{2}$$

It then becomes clear that if we multiply the transformed input matrix A_{im2col} and the flattened kernel matrix U_{im2col}, i.e., $D_{im2col} = A_{im2col} \times U_{im2col}$, D_{im2col} is exactly the flattened convolution output $D = conv_{2D}(A, U)$, as Eq. 3 shows.

$$\begin{aligned} D_{im2col} &= A_{im2col} \times U_{im2col} \\ &= \begin{pmatrix} d_{0,0} & d_{0,1} & d_{0,2} & d_{1,0} & d_{1,1} & d_{1,2} & d_{2,0} & d_{2,1} & d_{2,2} \end{pmatrix}^T \\ &= flattened(conv_{2D}(A, U)) \end{aligned} \tag{3}$$

Our idea started with a curious exploration to represent the convolution in a polynomial form. It is based on the known technique that matrices and vectors can be encoded as polynomials and their multiplication operations can be formulated as the multiplication between polynomials [12]. Our main challenge along this route is what kind of polynomials we should construct for the input matrix and the kernel vector in convolution.

We start with a pretty plain polynomial construction for the input matrix A. For the element $A_{i,j}$, it will be multiplied with t^k, where t is an indeterminate symbol representing no particular value and $k = i \times I_w + j$. In other words, the input matrix element $A_{i,j}$ becomes the coefficient of the k^{th} power of t. Note that each pair $[i, j]$ uniquely corresponds to a specific k, and vice versa. Therefore, the polynomial representation of the input matrix in the example can be depicted as Eq. 4.

$$\begin{aligned}A(t) = {} & a_{0,0}t^0 + a_{0,1}t^1 + a_{0,2}t^2 + a_{0,3}t^3 + a_{0,4}t^4 \\ & + a_{1,0}t^5 + a_{1,1}t^6 + a_{1,2}t^7 + a_{1,3}t^8 + a_{1,4}t^9 \\ & + a_{2,0}t^{10} + a_{2,1}t^{11} + a_{2,2}t^{12} + a_{2,3}t^{13} + a_{2,4}t^{14} \\ & + a_{3,0}t^{15} + a_{3,1}t^{16} + a_{3,2}t^{17} + a_{3,3}t^{18} + a_{3,4}t^{19} \\ & + a_{4,0}t^{20} + a_{4,1}t^{21} + a_{4,2}t^{22} + a_{4,3}t^{23} + a_{4,4}t^{24}\end{aligned} \tag{4}$$

The next question then becomes how we should construct the polynomial form for the kernel matrix or vector. The goal for the construction is that when we multiply the input polynomial and the kernel polynomial, the coefficients in the resultant polynomial are exactly the result of applying convolution between the input and the kernel.

The *im2col* process provides a proper conceptual basis for solving this problem. Based on the polynomial representation of A in Eq. 4, we can first construct the polynomial form for the *im2col* matrix A_{im2col} by multiplying the input elements of A_{im2col} in Eq. 1 with their corresponding degrees of the indeterminate t in Eq. 4. Therefore, the polynomial form of A_{im2col}, named A^t_{im2col}, can be defined as in the following Eq. 5, where $\odot$ is the element-wise product of matrices.

$$A^t_{im2col} = A_{im2col} \odot \begin{pmatrix} \begin{pmatrix} t^0 & t^1 & t^2 \\ t^1 & t^2 & t^3 \\ t^2 & t^3 & t^4 \end{pmatrix} & \begin{pmatrix} t^5 & t^6 & t^7 \\ t^6 & t^7 & t^8 \\ t^7 & t^8 & t^9 \end{pmatrix} & \begin{pmatrix} t^{10} & t^{11} & t^{12} \\ t^{11} & t^{12} & t^{13} \\ t^{12} & t^{13} & t^{14} \end{pmatrix} \\ \begin{pmatrix} t^5 & t^6 & t^7 \\ t^6 & t^7 & t^8 \\ t^7 & t^8 & t^9 \end{pmatrix} & \begin{pmatrix} t^{10} & t^{11} & t^{12} \\ t^{11} & t^{12} & t^{13} \\ t^{12} & t^{13} & t^{14} \end{pmatrix} & \begin{pmatrix} t^{15} & t^{16} & t^{17} \\ t^{16} & t^{17} & t^{18} \\ t^{17} & t^{18} & t^{19} \end{pmatrix} \\ \begin{pmatrix} t^{10} & t^{11} & t^{12} \\ t^{11} & t^{12} & t^{13} \\ t^{12} & t^{13} & t^{14} \end{pmatrix} & \begin{pmatrix} t^{15} & t^{16} & t^{17} \\ t^{16} & t^{17} & t^{18} \\ t^{17} & t^{18} & t^{19} \end{pmatrix} & \begin{pmatrix} t^{20} & t^{21} & t^{22} \\ t^{21} & t^{22} & t^{23} \\ t^{22} & t^{23} & t^{24} \end{pmatrix} \end{pmatrix} \tag{5}$$

We already know that $A_{im2col} \times U_{im2col}$ equals the convolution between A and U. Therefore, for the construction of the polynomial form of the kernel vector U_{im2col}, we want the degree of t in the result polynomial for the inner product between each row of A_{im2col} and U_{im2col} to be the same and different rows of A_{im2col} will produce different degrees. In this way, the coefficients in the result polynomial will be exact the result of convolution. For U, we can similarly construct U_t in a corresponding manner. Let us explain this desired property using the first two rows of A^t_{im2col} in our example. The first two rows are:

$$row_0 = (\, a_{0,0}t^0 \ a_{0,1}t^1 \ a_{0,2}t^2 \ a_{1,0}t^5 \ a_{1,1}t^6 \ a_{1,2}t^7 \ a_{2,0}t^{10} \ a_{2,1}t^{11} \ a_{2,2}t^{12} \,)$$

$$row_1 = (\, a_{0,1}t^1 \ a_{0,2}t^2 \ a_{0,3}t^3 \ a_{1,1}t^6 \ a_{1,2}t^7 \ a_{1,3}t^8 \ a_{2,1}t^{11} \ a_{2,2}t^{12} \ a_{2,3}t^{13} \,)$$

A general polynomial form of U_{im2col} can be written as:

$$(\, u_{0,0}t^{e0} \ u_{0,1}t^{e1} \ u_{0,2}t^{e2} \ u_{1,0}t^{e3} \ u_{1,1}t^{e4} \ u_{1,2}t^{e5} \ u_{2,0}t^{e6} \ u_{2,1}t^{e7} \ u_{2,2}t^{e8} \,)^T$$

$e0 \dots e8$ are the degrees of the polynomial terms. However, an arbitrary polynomial form won't work, in other words, the degrees in the polynomial form of U_{im2col}, i.e., $e0 \dots e8$ in this example, must be carefully selected. Assuming we already had a desired polynomial for U_{im2col} named U^t_{im2col}, what we want then is that $row_0 \cdot U^t_{im2col} = d_{0,0}t^i$, and $row_1 \cdot U^t_{im2col} = d_{0,1}t^j$. Importantly, the degrees i and j can be any values but must be different from each other, i.e., $i \neq j$. More broadly speaking, we want the inner product between each row of A^t_{im2col} and the to-be-constructed U^t_{im2col} produce only a single polynomial term, i.e., $coefficient \times t^i$, and the degrees of the term i are different for different rows. Therefore, the coefficients of all such resultant polynomial terms will be exactly the result of convolution.

There are more than one polynomial form of U_{im2col} that can satisfy the requirements. Let us look at two U^t_{im2col}'s that can both work for the calculation of convolution in our example. Then we will describe the general way to construct the polynomial for the kernel vector.

$$U^t_{im2col} = (\, u_{0,0}t^{12} \ u_{0,1}t^{11} \ u_{0,2}t^{10} \ u_{1,0}t^7 \ u_{1,1}t^6 \ u_{1,2}t^5 \ u_{2,0}t^2 \ u_{2,1}t^1 \ u_{2,2}t^0 \,)^T \tag{6}$$

If we conduct the polynomial multiplication between A^t_{im2col} and U^t_{im2col}, we get D_t

$$\begin{aligned} D_t = A^t_{im2col} \times U^t_{im2col} = \\ (\, d_{0,0}t^{12} \ d_{0,1}t^{13} \ d_{0,2}t^{14} \ d_{1,0}t^{17} \ d_{1,1}t^{18} \ d_{1,2}t^{19} \ d_{2,0}t^{22} \ d_{2,1}t^{23} \ d_{2,2}t^{24} \,)^T \end{aligned} \tag{7}$$

Upon examining the result of the polynomial multiplication, it becomes apparent that for any given row of A^t_{im2col}, when it is multiplied by U^t_{im2col}, the power of t in each resultant element is the same. This allows for their addition into a single polynomial term for a row. Moreover, in the resulting vector D_t, the power of t in each element is unique. And importantly, the coefficients of all the different polynomial terms in D_t are exactly the same element as in the vector $flattened(conv_{2D}(A, U))$.

Let us look at an alternative polynomial form for U_{im2col}:

$$\begin{aligned} alternative\ U^t_{im2col} = \\ (\, u_{0,0}t^{19} \ u_{0,1}t^{18} \ u_{0,2}t^{17} \ u_{1,0}t^{14} \ u_{1,1}t^{13} \ u_{1,2}t^{12} \ u_{2,0}t^9 \ u_{2,1}t^8 \ u_{2,2}t^7 \,)^T \end{aligned} \tag{8}$$

And we get the corresponding D_t

$$alternative\ D_t = A^t_{im2col} \times alternative\ U^t_{im2col} = \begin{pmatrix} d_{0,0}t^{19} & d_{0,1}t^{20} & d_{0,2}t^{21} & d_{1,0}t^{24} & d_{1,1}t^{25} & d_{1,2}t^{26} & d_{2,0}t^{29} & d_{2,1}t^{30} & d_{2,2}t^{31} \end{pmatrix}^T \quad (9)$$

The result polynomial is different. However, its coefficients are the same convolution vector that we want to calculate.

A general way to construct U^t_{im2col} can be derived from the characteristic of doubly Hankel matrices. This specific type of matrix has two layers of diagonals that are congruent. A property derived from this two layer congruence is that the vector of the elements' linear index in each row is mirror symmetric to the reverse of the same vector. This property is the core of how we construct the kernel vector polynomial. Let us still use the same example to illustrate how this property is useful.

If we pay attention to the first row of A^t_{im2col} in Eq. 5, it is

$$\begin{pmatrix} a_{0,0}t^0 & a_{0,1}t^1 & a_{0,2}t^2 & a_{1,0}t^5 & a_{1,1}t^6 & a_{1,2}t^7 & a_{2,0}t^{10} & a_{2,1}t^{11} & a_{2,2}t^{12} \end{pmatrix}$$

If we extract the polynomial degree of t into a vector it is

$$\overrightarrow{RD_{1st}} = \overrightarrow{first\_row\_degree} = \begin{pmatrix} 0 & 1 & 2 & 5 & 6 & 7 & 10 & 11 & 12 \end{pmatrix}$$

The elements in the vector are actually also the flattened index of the corresponding elements $\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{1,0} & a_{1,1} & a_{1,2} & a_{2,0} & a_{2,1} & a_{2,2} \end{pmatrix}$ Very importantly,

$$\overrightarrow{RD_{1st}} + reverse(\overrightarrow{RD_{1st}}) = \begin{pmatrix} 12 & 12 & 12 & 12 & 12 & 12 & 12 & 12 & 12 \end{pmatrix}$$

The sums 12 are all the same, and is actually the last value in the row vector. Furthermore, when the reverse vector of the first row's degrees is added with the vector of degrees in other rows, the sums are also the same for all other rows. For example,

$$\overrightarrow{RD_{2nd}} = \overrightarrow{2nd\_row\_degree} = \begin{pmatrix} 1 & 2 & 3 & 6 & 7 & 8 & 11 & 12 & 13 \end{pmatrix}$$

and

$$\overrightarrow{RD_{2nd}} + reverse(\overrightarrow{RD_{1st}}) = \begin{pmatrix} 13 & 13 & 13 & 13 & 13 & 13 & 13 & 13 & 13 \end{pmatrix}$$

The sum 13 is also the last value in the second row vector.

This property leads to a general way to construct the polynomial form for U_{im2col} as the element-wise product of U_{im2col} and $reverse(\overrightarrow{row\_degree})$, i.e.,

$$U^t_{im2col} = U_{im2col} \odot reverse(\overrightarrow{row\_degree})$$

Actually the first kernel polynomial construction example in Eq. 6 exactly comes from here.

We don't need to exclusively use the first row for the construction. All row vectors will work, just like how the alternative U^t_{im2col} is constructed in Eq. 8. However, constructing with the first row is more efficiently to implement because it simplifies the identification of the polynomial terms whose coefficients are the result vector of convolution. If we use the first row vector, the degrees of the relevant polynomial terms will be the last column of A^t_{im2col}. Let p_i represent the coefficient of t^i in $P(t)$. In this example, $d_{0,0} = p_{12}$, $d_{0,1} = p_{13}$, $d_{0,2} = p_{14}$, $d_{1,0} = p_{17}$, $d_{1,1} = p_{18}$, $d_{1,2} = p_{19}$, $d_{2,0} = p_{22}$, $d_{2,1} = p_{23}$, $d_{2,2} = p_{24}$.

The vector of degrees of these terms $\begin{pmatrix} 12 & 13 & 14 & 17 & 18 & 19 & 22 & 23 & 24 \end{pmatrix}$ is exactly the degrees of the last column of A^t_{im2col} in Eq. 5. From here on, we will only use the construction with the first row vector.

With the construction of A^t_{im2col} and U^t_{im2col}, we successfully transform the calculation of convolution into a polynomial multiplication problem. Now we can generalize the construction to any size of input and kernel in neural networks. Conceptually the *im2col* matrix of the problem is a doubly blocked Hankel matrix A, consisting of $O_h \times K_h$ blocks, each of size $O_w \times K_w$. The vector U, which is used for multiplication, has a size of $K_h \times K_w$, and the resultant vector D has a size of $O_h \times O_w$. Based on this, we construct the polynomials

$$A(t) = \sum_{i=0}^{O_h+K_h-2} \sum_{j=0}^{O_w+K_w-2} a_{i,j} t^{(O_w+K_w-1)\times i+j} \quad (10)$$

and

$$U(t) = \sum_{i=0}^{K_h-1} \sum_{j=0}^{K_w-1} u_{i,j} t^{-(O_w+K_w-1)\times i-j+(Ow+K_w-1)\times K_h-O_h-1} \quad (11)$$

Then, we calculate $P(t) = A(t) \times U(t)$. Finally, within the coefficients of $P(t)$, we construct the resultant vector D. Let p_i be the coefficient of t^i in P

$$\begin{aligned} d_{i,j} =& p_{(O_w+K_w-1)\times i+j+(O_w+K_w-1)\times K_h-O_w} \\ & 0 \le i < O_h, 0 \le j < O_w \end{aligned} \quad (12)$$

We want to emphasize that as Eq.s 10, 11 and 12 clearly show, the input polynomial $A(t)$ and the kernel polynomial $U(t)$ can be calculated directly from the input A and the kernel U. Their calculation is not dependent on the prerequisite that the input need to be transformed by the *im2col* process first. The *im2col* matrix transformed from input is only used as a conceptual tool that facilitates the deduction of the formulas. Therefore, our approach doesn't incur the data redundancy that *im2col* introduces.

2.3 Polynomial Multiplication Using FFT

After we transform the convolution problem in the form of multiplying a doubly Hankel matrix with a vector into a polynomial multiplication problem, we can use FFT to solve it. Solving polynomial multiplication with FFT is a well know and broadly used algorithmic technique. Here we briefly overview the general FFT-based solution, and the specifics when we apply the general technique to our convolution-derived polynomial multiplication problem.

Suppose we have the following two generic polynomials, $P(t)$ and $Q(t)$, where $P(t)$ has a total of N terms and Q has M terms:

$$P(t) = \sum_{i=0}^{N-1} p_i t^i \qquad Q(t) = \sum_{i=0}^{M-1} q_i t^i \quad (13)$$

The goal of the polynomial multiplication problem is to calculate $R(t) = P(t)Q(t)$, more specifically, figuring out all the terms in $R(t)$ including the coefficients and the degrees of the terms.

There is a naive method for solving the problem, which involves multiplying each term of P with every term of $Q(t)$. The time complexity of this naive method is $O(MN)$. However, it is known that FFT can be used to solve this problem. The FFT based method reduces the time complexity to $O((M + N)\log(M + N))$. The theoretical process of the FFT method needs to evaluate the $P(t)$ and $Q(t)$ polynomials over the complex roots of unity.

In our method, the two polynomials are the $A(t)$ and $U(t)$ defined in Eq.s 10 and 11, respectively, i.e., $P(t) = A(t)$ and $Q(t) = U(t)$. We start with constructing two vectors, P and Q, indexed by the powers of t in $P(t)$ and $Q(t)$ respectively, with the coefficients of the t terms as their elements. P and Q are also called coefficient vector form in literature. Subsequently, we pad the end of vector P with $M - 1$ zeros and the end of vector Q with $N - 1$ zeros, as follows

$$\begin{aligned} P &= [p_0, p_1, \ldots, p_{N-2}, p_{N-1}, \underbrace{0, \ldots, 0}_{\text{M-1 zeros}}] \\ Q &= [q_0, q_1, \ldots, q_{M-2}, q_{M-1}, \underbrace{0, \ldots, 0}_{\text{N-1 zeros}}] \end{aligned} \tag{14}$$

Afterwards, we perform FFT operations on both P and Q, obtaining $\hat{P}$ and $\hat{Q}$ respectively. We then carry out a pointwise multiplication of $\hat{P}$ with $\hat{Q}$, and perform an Inverse FFT operation on the result to obtain the result vector R. The elements of R are the coefficients of the resultant polynomial $R(t)$, with their indices representing the powers of t. The process can be summarized as follows:

$$R = \text{IFFT}(\text{FFT}(P) \cdot \text{FFT}(Q)) \tag{15}$$

Then the coefficients of the terms with the degrees defined in Eq. 12 are the result of the convolution.

2.4 Complexity Analysis

Next we will analyze the complexity of our approach and also that of the *im2col+MM* method and the traditional *FFT* method. We will analyze both the operational complexity, which is the number of operations, and also the space complexity, which is the size of data storage or the number of memory transfer needed, in the methods.

The complexity analysis is done on each step of all the methods. The *im2col* method conducts matrix-vector multiplications on conceptually sliding-duplicated matrices and the flattened kernel. The *FFT* method conducts essentially a 2D-FFT on the padded input, a 2D-FFT the padded kernel, an element-wise multiplication, and a 2D-IFFT. PolyHankel does two 1D-FFTs, one 1D-IFFT, and one element-wise multiplication based on the constructed input and kernel polynomials.

Table 2 and Table 3 list the time- and space-complexity of the three methods. Typically $I_{h/w} \gg K_{h/w}$, so $O_{h/w} \approx I_{h/w}$. It shows that our PolyHankel method has lower operational and space complexity than FFT, and requires much smaller extra memory overhead (either storage or transfers) compared with the $im2col + MM$ method.

A place worth noting is that both the FFT-based method and PolyHankel use padding. The complexity expressions include the impact of padding. Specifically, both input and kernel are padded to $I_h * I_w + K_h * I_w$. That's why the term $I_h * I_w + K_h * I_w$ appears repeatedly in the complexity expression, as it is the size of the padded FFT.

3 Implementation

In this section, we describe how we adapt the implementation for additional NN parameters and computer system factors.

3.1 Calculating The Degrees of Polynomial Terms

A key task in our approach is to calculate the degrees of the relevant polynomial terms. The degrees are defined in Eq.s 10, 11 and 12. If we strictly follow the equations, the calculation will not be efficient. In our practical implementation, we use a mapping method to generate the powers of t, which allows for a more intuitive construction of polynomials. First, we need to establish a map to associate each unique element in the doubly blocked Hankel matrix with a value. Our map starts with the element value at zero and incrementally increases by one as we traverse to the next element. The doubly blocked Hankel matrix has a two-tier structure. For the outer layer, we traverse each block of the first row from left to right, and then each block of the rightmost column from top to bottom, forming a L-shaped path. Within each block, the path is similar, i.e., first traversing all elements of the first row from left to right, then each element of the rightmost column from top to bottom. After completing the traversal, we have mapped an integer to each unique value in the doubly blocked Hankel matrix. Using the same example in Section 2, our map is illustrated in Fig. 2.

$$\begin{pmatrix} \begin{pmatrix} 0_* & 1_* & 2_* \\ & & 3 \\ & & 4 \end{pmatrix} & \begin{pmatrix} 5_* & 6_* & 7_* \\ & & 8 \\ & & 9 \end{pmatrix} & \begin{pmatrix} 10_* & 11_* & \mathbf{12}_* \\ & & \mathbf{13} \\ & & \mathbf{14} \end{pmatrix} \\ \begin{pmatrix} \ldots \\ \ldots \\ \ldots \end{pmatrix} & \begin{pmatrix} \ldots \\ \ldots \\ \ldots \end{pmatrix} & \begin{pmatrix} 15 & 16 & \mathbf{17} \\ & & \mathbf{18} \\ & & \mathbf{19} \end{pmatrix} \\ \begin{pmatrix} \ldots \\ \ldots \\ \ldots \end{pmatrix} & \begin{pmatrix} \ldots \\ \ldots \\ \ldots \end{pmatrix} & \begin{pmatrix} 20 & 21 & \mathbf{22} \\ & & \mathbf{23} \\ & & \mathbf{24} \end{pmatrix} \end{pmatrix}$$

Figure 2. Indices corresponding to the degrees of the input polynomial. Starred elements are the reverse of the indices for the kernel polynomial. Bold elements are the indices for the result polynomial.

Since the antidiagonal blocks of a doubly Hankel matrix are identical, and within each block, the elements on the antidiagonals are also identical, the positions of the elements in this map encompass all the non-repetitive elements of

Table 2. Time Complexity Analysis

Method	Time Complexity
im2col + MM	$\underbrace{K_h \times K_w \times O_h \times O_w}_{\text{matrix-vector multiplication}}$
Traditional FFT	$\underbrace{(I_w+K_w)(I_h+K_h)(\log(I_h+K_h)+\log(I_w+K_w))\times 2}_{\text{FFT on input and kernel}} + \underbrace{(I_h+K_h)*(I_w+K_w)}_{\text{element-wise multiplication}} + \underbrace{(I_w+K_w)(I_h+K_h)(\log(I_h+K_h)+\log(I_w+K_w))}_{\text{IFFT}}$
Fine-grain FFT	$\underbrace{I_h \times 2I_w \log(2I_w)}_{\text{FFT on input blocks}} + \underbrace{K_h \times 2I_w \log(2I_w)}_{\text{FFT on kernel blocks}} + \underbrace{(O_h \times K_h \times I_w)}_{\text{element-wise multiplication}} + \underbrace{(O_h \times 2I_w log(2I_w))}_{\text{IFFT}}$
Our Poly-Hankel Method	$\underbrace{(I_h \times I_w + K_h \times I_w)\log(I_h \times I_w + K_h \times I_w)}_{\text{FFT on input polynomial}} + \underbrace{(I_h \times I_w + K_h \times I_w)\log(I_h \times I_w + K_h \times I_w)}_{\text{FFT on kernel polynomial}} + \underbrace{(I_h \times I_w + K_h \times I_w)}_{\text{element-wise multiplication}} + \underbrace{(I_h \times I_w + K_h \times I_w)\log(I_h \times I_w + K_h \times I_w)}_{\text{IFFT}}$

Table 3. Space Complexity Analysis

Method	Space Complexity
im2col + MM	$\underbrace{K_h \times K_w \times O_h \times O_w}_{\text{im2col expanded matrix}}$
Traditional FFT	$\underbrace{(I_h+K_h)(I_w+K_w)}_{\text{padded input for FFT}} + \underbrace{(I_h+K_h)(I_w+K_w)}_{\text{padded kernel for FFT}} + \underbrace{(I_h+K_h)(I_w+K_w)}_{\text{output of element-wise multiplication}}$
Fine-grain FFT	$\underbrace{(I_h \times 2I_w)}_{\text{FFT result of input blocks}} + \underbrace{(K_h \times 2I_w)}_{\text{FFT result of kernel blocks}} + \underbrace{(O_h \times 2I_w)}_{\text{output of element-wise multiplication}}$
Our Poly-Hankel Method	$\underbrace{(I_h \times I_w + K_h \times I_w)}_{\text{padded input polynomial}} + \underbrace{(I_h \times I_w + K_h \times I_w)}_{\text{padded kernel polynomial}} + \underbrace{(I_h \times I_w + K_h \times I_w)}_{\text{output of element-wise multiplication}}$

the doubly Hankel matrix A. To construct the polynomial $U(t)$, we take elements in the map as the powers of t and multiply them by the corresponding elements of A. In order to construct the polynomial $A(t)$, we take all elements in the map to represent powers of t, thereby constructing terms of t. Subsequently, the corresponding elements from A in the map are used as coefficients for these terms. By summing all these terms together, we obtain the polynomial $A(t)$. Afterwards, we use the map to construct the polynomial $U(t)$. We extract the first row of the map, referred to as a vector, and perform an inverse operation on it. Similarly, we then use the elements at the corresponding positions of this vector as powers of t to construct terms of t. The corresponding elements of U at these positions are used as coefficients for these t terms. We sum all these terms to obtain the polynomial $U(t)$. Finally, we multiply $A(t)$ with $U(t)$ to obtain the resulting polynomial $D(t)$. Additionally, it is necessary to extract coefficients from the polynomial $D(t)$ to construct the final result vector D. We extract the rightmost column of the map and treat it as a vector. Using the elements of this vector as indices, we identify terms in $D(t)$ where the power of t matches these indices. We then extract the coefficients of these terms and, following the order of indices in the vector, construct a new vector. This is the result vector D.

3.2 Network Channels

The technique described so far works for the convolution induced in single-channel neural networks. The rearrangement of the input and kernel so far are also made for this assumption. Neural networks normally have multiple channels. We have two viable options to support multiple channels. The first option involves merging all input channels and then performing FFT on the merged channels, as our algorithm naturally aggregates the outcomes of different channels. However, this approach increases the FFT size. The alternative is to execute the FFT on each input channel individually and subsequently sum their outputs. The computational complexity of the former approach is $O(CI_hI_w\log(CI_hI_w) + CK_hI_w\log(CK_hI_w))$, The latter has a complexity of $O(CI_hI_w\log(I_hI_w) + CK_hI_w\log(K_hI_w) + CO_h)$. Our experimentation reveals that an increase in input size significantly increases the execution time for FFT, surpassing the time needed for summing different channels. Consequently, we opt for the second method, performing FFT on separate channels and adding the outputs during element-wise multiplication.

Typically, convolution operations require padding. Thus, at this stage, zero-padding is applied at the beginning and the end of the input as per specific requirements. Moreover, given our adoption of the overlap-save technique for optimization, additional zero-padding at the start and end of each batch is essential to meet the overlap-save criteria. More specifically, the handling of input is the same for multi-channel as for single channel. The handling of kernel, however, needs to be adapted for the multiple channel scenario. The adaption is actually quite simple—we just need to use non-overlapping degrees in the kernel polynomial U_t defined in Eq. 11 for kernels in different channels. The first step is to reverse the position of each element within the kernel. Subsequently, as depicted in Figure 2, we rearrange the kernel. Then because the kernel polynomials for different channels are not overlapping, these polynomials can be merged in to a combined kernel polynomial.

The combined kernel size should be: KernelSize = $(K_h - 1) \times I_w + K_w$. Each row of the kernel necessitates padding with $I_w - K_w$ zeros. In this step, the last row does not require additional padding. Subsequently, we employ cuFFT to perform FFT on both input and kernel. Since cuFFT is highly optimized and perform best for FFT sizes satisfying $2^a \times 3^b \times 5^c \times 7^d$, further padding at the end is mandated to meet cuFFT's criteria. Our tests indicate that FFT sizes as multiples of 2 exhibit optimal performance. Hence, we pad the kernel size to the nearest multiple of 2.

Thereafter, the same element-wise multiplication between the input polynomial and the merged kernel polynomials is conducted, followed by the summation of outputs across different channels, as well as the IFFT on the result. Again because the relevant degrees constructed for different channels are not overlapping, the result can be naturally dissected into corresponding channels. The final step is different for multi-channel. We select the corresponding IFFT values and copy them into the result array. For each batch, we disregard the first $(K_h - 1) \times I_w + K_w - 1$ values, selecting the subsequent O_w values. Then, we skip the next $K_w - 1$ values, continuing this pattern for the remaining output. It is crucial to account for the index offsets caused by overlap-save and zero-padding during index calculation.

4 Evaluation and Performance Analysis

We evaluate the proposed convolution method from three aspects: (1) API-level performance comparison with NVIDIA's cuDNN library [4] and Zhang's Fine-grain FFT method, (2) application-scenario evaluation with networks developed in PyTorch[18], a leading deep learning programming framework, by replacing PyTorch's invocation of cuDNN for convolution with our own implementation, and (3) performance profiling to analyze and understand the source of performance improvement. Our method is referred as PolyHankel.

Each data point is the average of ten runs. The runs are typically quite stable with variance at around 3%. Since the performance of convolution is independent to input values, we randomly generate inputs and use the same input for each data point. The versions of cuDNN and PyTorch are 8.9 and 2.1, respectively. Specifically, cuDNN has multiple variants of the GEMM method, including the original GEMM method, the Implicit method and the Implicit Precomputed method [15]. In our experiment, all the measurements with regard to the GEMM method, both for performance comparison and for profiling, are based on cuDNN's IMPLICIT_PRECOMP_GEMM algorithm flag, because it is typically the fastest variant within the GEMM algorithm family.

we evaluate on three Nvidia GPUs, GeForce 3090Ti, A10G and V100. In our experiments, CPU only serves as the command processor and has a negligible impact on performance.

4.1 API Performance Comparison

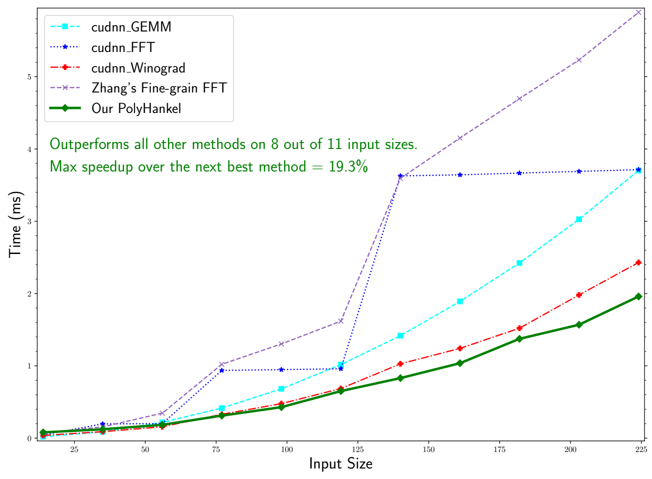
We compare PolyHankel with four other state-of-the-art convolution methods, three from cuDNN and the other being Zhang' recent Fine-grain FFT method. cuDNN is NVidia's own NN library that is deeply optimized and provide some of the fastest convolution implementations. The three cuDNN methods we compare to are $im2col + GEMM$, FFT and $Winograd$, which are the more frequently used convolution methods. We use the same API design in PolyHankel as that in cuDNN. We measure and compare the end-to-end API execution time.

The performance of convolution is most sensitive to input sizes and kernel sizes. As mentioned in Section 1, no method can outperform others in all cases. A normal practice is that developers will choose different convolution methods for different cases. Therefore, we vary these parameters to measure the strengths and weaknesses of PolyHankel in different parts of the parameter space. The experiment is then categorized into groups. Each group fixes the values of one parameter and varies the other one. Thus, we can study how this parameter impact the overall performance of the algorithm. Notably the performance of convolution is insensitive to the value of inputs and weights. Therefore, the performance we observe at the API level is going to be consistent with that in real-world networks.

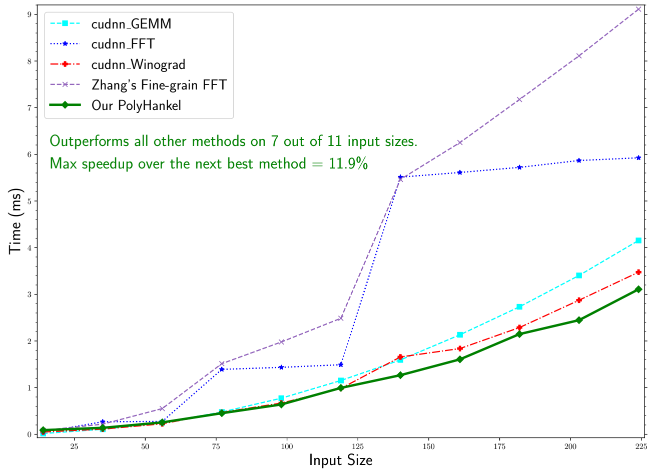
Performance vs. input sizes: Figure 3 shows the performance of the three methods over input sizes from 4 to 224. In this group of experiments, the kernel size is set to 5, a widely used kernel size configuration, and the batch size is 128. Generally PolyHankel outperforms all other methods for sizes larger than 100. On the three types of GPU, PolyHankel achieves the maximal speedups over the next best method of 19.3%, 11.9%, 48.9%, respectively.

Performance vs. kernel sizes: Figure 4 shows the performance of the methods over kernel sizes from 4 to 25. The Winograd method has only one data point in the experiment because cuDNN only support one kernel size for this method. PolyHankel has notable speedups over all other methods for kernel sizes < 15, with the maximal speedups over the next best method on the three GPUs being 34.6%, 43.1%, 33.6%, respectively. In NN applications, the typical choices of kernel size are between 3 to 9, with 3×3 and 5×5 being the most common. So PolyHankel will have advantages in common application scenarios.

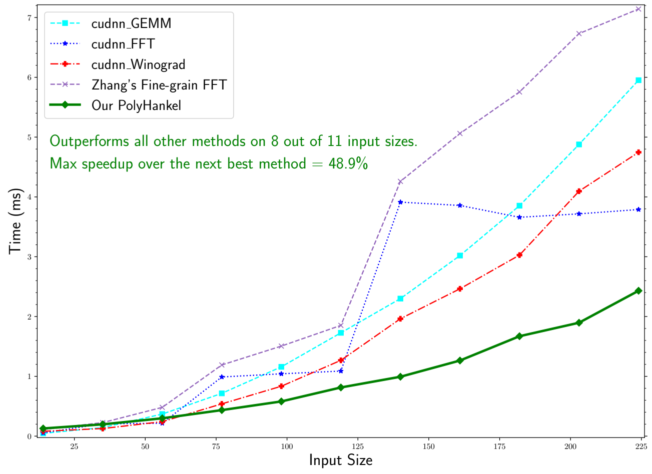
Notably, The FFT convolution tends to be constant and insensitive to the kernel size, because it zero-pads the kernel to be the same size as the input image and kernel size has nearly no effect on the performance. In contrast, the performance of our method decreases with larger kernel sizes. This is because the FFT size in PolyHankel is determined by the



(a) GeForce 3090Ti

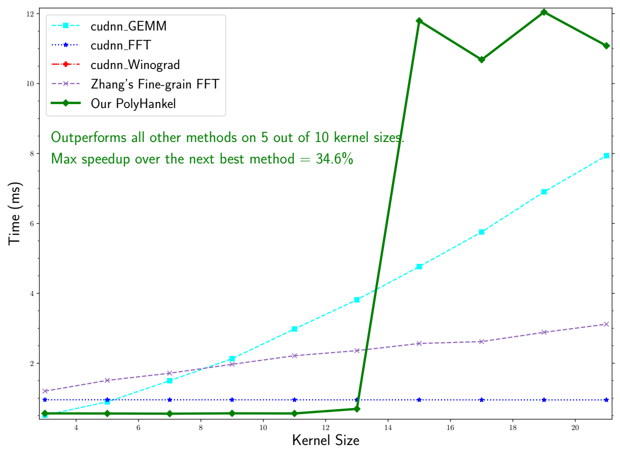


(b) A10G

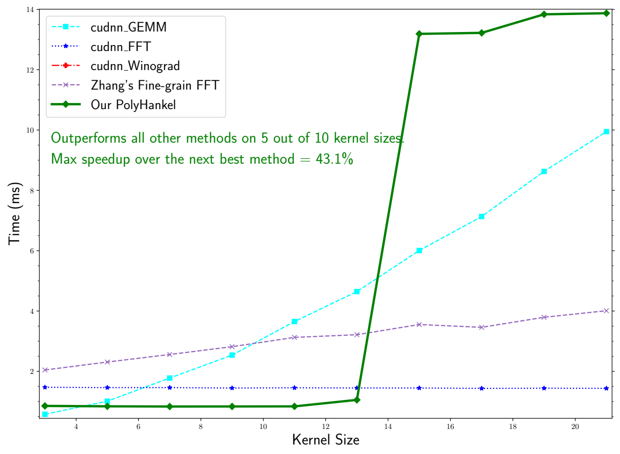


(c) V100

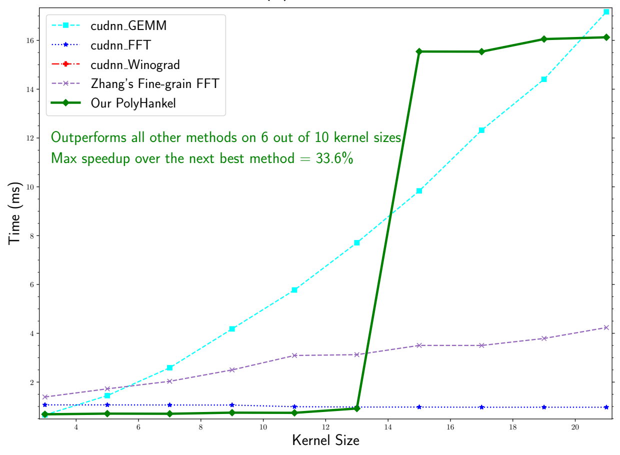
Figure 3. API Performance Comparison on Different Input Sizes



(a) GeForce 3090Ti



(b) A10G



(c) V100

Figure 4. API Performance Comparison on Different Kernel Sizes. Here the Winograd method has only one data point, because cuDNN only supports the kernel size 3 for the Winograd method.

size of kernel vectors. When the kernel vector size reaches the next power of two, the FFT size will be doubled.

The *im2col + GEMM* method loses performance as the kernel size increases, mainly because it fully unrolls matrices, and the matrix sizes grow quadratically with the kernel size.

However, when the kernel size is small, *im2col + GEMM* has better performance over the FFT method because the unrolled matrix is small and the high performance of matrix multiplication routine in cuDNN outweighs the saving of algorithmic complexity in the FFT method.

Performance vs. channel counts: Figure 5 shows the performance comparison of our method over all cuDNN's methods, including GEMM and its variants, FFT and its tiled variant, Winograd and its nonfused variant. The channel count changes from 1 to 128. The setup of other parameters is input size = 112∗112, kernel size = 3∗3. Generally PolyHankel outperforms all cuDNN's methods. Particularly notable is that the cuDNN's methods show quite diverse performance trends. In other words, there is no single cuDNN method performing best across all channel counts.

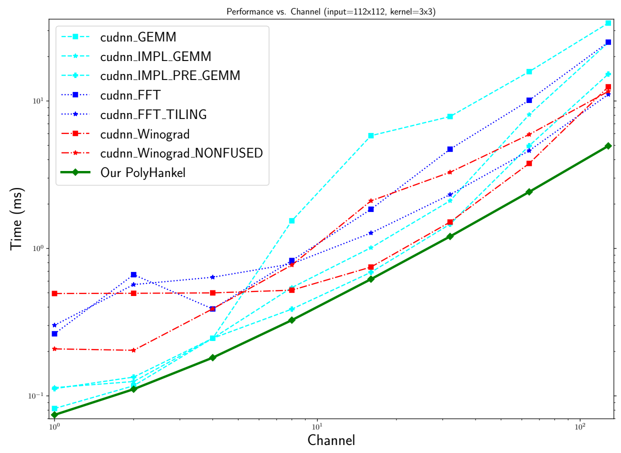
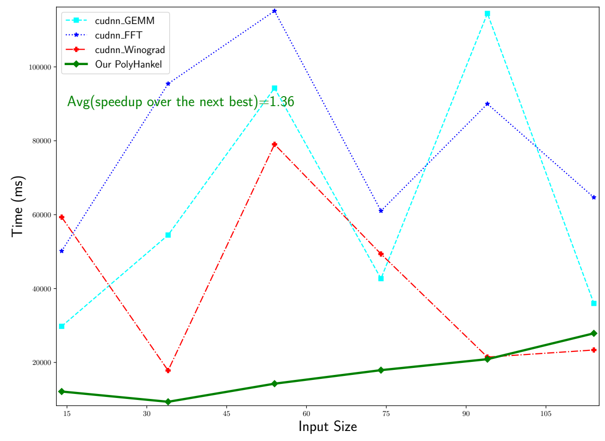


Figure 5. API Performance Comparison on Different Channel Counts. The comrison is conducted against all cuDNN's methods with input size=112, kernel=3 and channel count changing from 1 to 128 on 3090Ti. Both axes are in log scale.

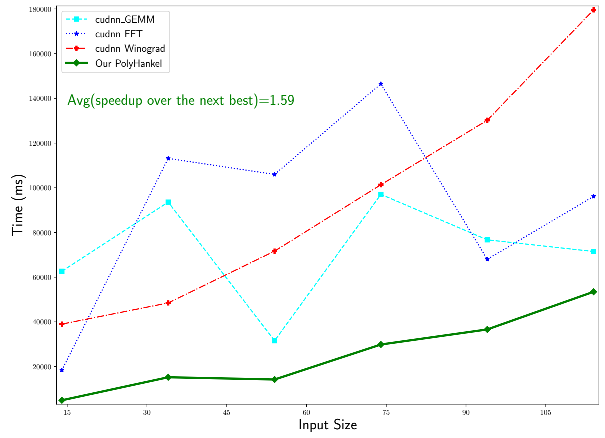
4.2 Performance with Neural Networks

We also evaluate PolyHankel's performance in neural networks. The evaluation is done in PyTorch, one of the most popular frameworks for developing deep-learning applications. In this experiment, we replace PyTorch's invocation to cuDNN for convolution with calls to our method that has almost the same interface design. We compare the performance before/after the replacement for a set of synthetic networks. All the networks have 20 layers but have various layer designs including connection configurations and kernel sizes. We modified the code of PyTorch version 2.1 for this experiment. The fine-grain FFT method is not used in this experiment because the provided code was developed for another NN development environment Caffe [9] and can't be ported to PyTorch without significant modifications.

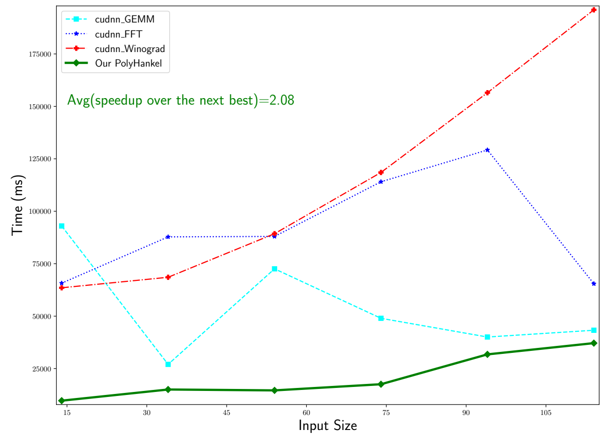
Figure 6 shows the execution time over input sizes varying from 4 to 224. On average, PolyHankel achieves the speedups over the next best method of 1.36, 1.59 and 2.08 on 3090Ti, A10G and V100, respectively. The data shows that PolyHankel maintain its performance advantage over the other cuDNN methods end-to-end in more realistic application scenarios.



(a) GeForce 3090Ti



(b) A10G



(c) V100

Figure 6. End-to-end Performance Comparison in PyTorch for Neural Networks. The data is the accumulated time spent on the convolution operator. Within a NN, convolution will be called with widely different parameter values such as input size, channel size, etc.

We want to specifically point out that the "fluctuations" in the performance are *not* the reflection of the noise in measurement. The fluctuation is caused by that one method is used through the whole network. In this experiment we force PyTorch to use one convolution method, and accumulate the time spent on that convolution operator. However, even for a simple network, convolution is called with different parameter values. For example, layer 1 might call with input size 112 and kernel size 3, but layer 2 will change to 56 and 5. Therefore even within one network, there can be places one method is better than others but lose in other places. Ideally, heuristics should be developed to choose the best convolution method for each API invocation.

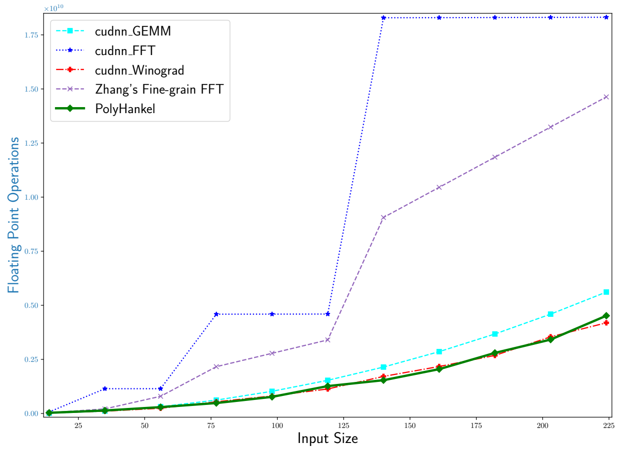
4.3 Performance Counter Profiling and Analysis

In order to explain the performance gain of PolyHankel, we profile all the methods with the GPU performance counters. The performance difference is mostly caused by the different number of memory transfers and floating point operations. Section 2.4 gives the theoretical analysis of these two metrics. In this experiment, we want to verify and correlate with the result of the theoretical analysis with NVidia CUDA's performance counters that record the total number of memory transactions and the number of floating point operations.

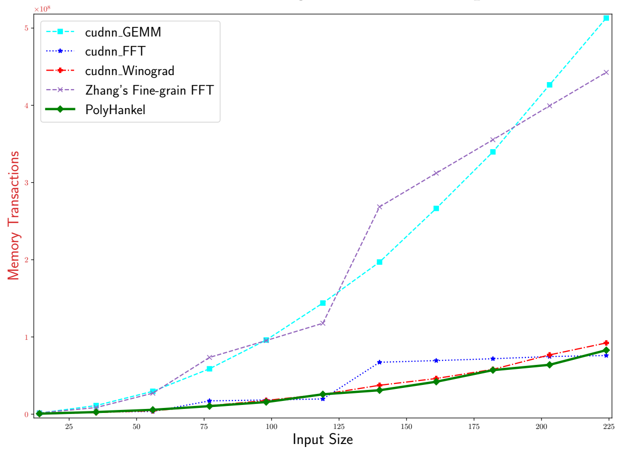
Figure 7 shows the performance counter profiles on A10G with input sizes varying from 4 to 224. We can see from the figure that the memory performance and the operational performance align well with the execution time on all the algorithms. It shows that PolyHankel generally has the lower number of operations and the lower number of memory transactions. In contrast, $im2col + MM$ has good operational efficiency but high memory overhead. cudnn_FFT is just the opposite, i.e., low number of memory transactions but high operational overhead. Winograd is good on both metrics, but has relatively higher memory overhead than PolyHankel, especially for larger input sizes. This experiment with performance counters explains why PolyHankel performs better over other method in a broad range of invocation scenarios is that it reaches a better performance tradeoff between the memory and operational efficiency.

5 Conclusion and Future Work

This paper presents a polynomial multiplication derived approach for solving convolution in neural networks. Our method constructs a conceptual polynomial for the im2col matrix and another polynomial for the kernel in a special way, so that convolution, originally in the form of the multiplication between the im2col matrix and the kernel, can be converted to a polynomial multiplication coefficient-finding problem. The polynomial coefficient problem can then be solved efficiently with FFT. We evaluate our methods on multiple application scenarios on three NVIDIA GPUs, and achieves significant speedups over broad input scenarios.



(a) Profiles of Floating Point Operations. PolyHankel typically has the lowest number of floating operations. The *im2col* (GEMM) and the Winograd methods also have low numbers of operations, but the *FFT* method has the highest number of operations.



(b) Profiles of Memory Transactions. PolyHankel typically has the lowest number of memory transactions. In contrast to the operational efficiency comparison, the *im2col* (GEMM) typically has the highest number of memory transactions , but the *FFT* and the Winograd methods have low number of transactions.

Figure 7. Profiling performance counters vs. input sizes.

One specific task for future work is to support tensor cores. Our method does not currently use tensor cores. We have found that whether or not our method is faster than cuDNN's methods that do use tensor cores is highly dependent on which kernel cuDNN dispatches, which in turn depends on the precision requirement, the microarchitecture, and the shape of the inputs. We believe our method could exploit tensor cores via the matrix form implementation of FFT and the elementwise multiplication, but for now this is future work.

References

[1] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. 2021. Computing graph neural networks: A survey from algorithms to accelerators. *ACM Computing Surveys (CSUR)* 54, 9 (2021), 1–38.

[2] Trinayan Baruah, Kaustubh Shivdikar, Shi Dong, Yifan Sun, Saiful A Mojumder, Kihoon Jung, José L Abellán, Yash Ukidave, Ajay Joshi, John Kim, and D. Kaeli. 2021. Gnnmark: A benchmark suite to characterize graph neural network training on gpus. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 13–23.

[3] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High Performance Convolutional Neural Networks for Document Processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*, Guy Lorette (Ed.). Université de Rennes 1, Suvisoft, La Baule (France). https://hal.inria.fr/inria-00112631 http://www.suvisoft.com.

[4] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). arXiv:1410.0759 http://arxiv.org/abs/1410.0759

[5] Jason Cong and Bingjun Xiao. 2014. Minimizing Computation in Convolutional Neural Networks. In *Artificial Neural Networks and Machine Learning – ICANN 2014*. Springer International Publishing, Cham, 281–290.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms* (2nd ed.). The MIT Press.

[7] Stefan Hadjis, Firas Abuzaid, Ce Zhang, and Christopher Ré. 2015. Caffe con Troll: Shallow Ideas to Speed Up Deep Learning. In *Proceedings of the Fourth Workshop on Data analytics in the Cloud, DanaC 2015, Melbourne, VIC, Australia, May 31 - June 4, 2015*, Asterios Katsifodimos (Ed.). ACM, 2:1–2:4. https://doi.org/10.1145/2799562.2799641

[8] Yangqing Jia. 2014. *Learning Semantic Image Representations at a Large Scale.* Ph. D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-93.html

[9] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the ACM International Conference on Multimedia, MM '14, Orlando, FL, USA, November 03 - 07, 2014*, Kien A. Hua, Yong Rui, Ralf Steinmetz, Alan Hanjalic, Apostol Natsev, and Wenwu Zhu (Eds.). ACM, 675–678. https://doi.org/10.1145/2647868.2654889

[10] Andrew Lavin and Scott Gray. 2016. Fast Algorithms for Convolutional Neural Networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 4013–4021. https://doi.org/10.1109/CVPR.2016.435

[11] Yann LeCun, Koray Kavukcuoglu, and Clément Farabet. 2010. Convolutional networks and applications in vision. In *International Symposium on Circuits and Systems (ISCAS 2010), May 30 - June 2, 2010, Paris, France*. IEEE, 253–256. https://doi.org/10.1109/ISCAS.2010.5537907

[12] David Lee. 1986. Fast multiplication of a recursive block Toeplitz matrix by a vector and its application. *Journal of Complexity* 2, 4 (1986), 295–305. https://doi.org/10.1016/0885-064X(86)90007-5

[13] Xiaqing Li, Guangyan Zhang, H. Howie Huang, Zhufan Wang, and Weimin Zheng. 2016. Performance Analysis of GPU-Based Convolutional Neural Networks. In *45th International Conference on Parallel Processing, ICPP 2016, Philadelphia, PA, USA, August 16-19, 2016*. IEEE Computer Society, 67–76. https://doi.org/10.1109/ICPP.2016.15

[14] Michaël Mathieu, Mikael Henaff, and Yann LeCun. 2014. Fast Training of Convolutional Networks through FFTs. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1312.5851

[15] NVIDIA. 2024. Implicit GEMM Convolution. https://github.com/NVIDIA/cutlass/blob/main/media/docs/implicit_gemm_convolution.md

[16] Hyunsun Park, Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. 2016. Zero and data reuse-aware fast convolution for deep neural networks on GPU. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES 2016, Pittsburgh, Pennsylvania, USA, October 1-7, 2016*. ACM, 33:1–33:10. https://doi.org/10.1145/2968456.2968476

[17] J.R. Partington. 1988. *An Introduction to Hankel Operators*. Cambridge University Press.

[18] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *CoRR* abs/1912.01703 (2019). arXiv:1912.01703 http://arxiv.org/abs/1912.01703

[19] Hugh Perkins. 2016. cltorch: a Hardware-Agnostic Backend for the Torch Deep Neural Network Library, Based on OpenCL. *CoRR* abs/1606.04884 (2016). arXiv:1606.04884 http://arxiv.org/abs/1606.04884

[20] Nicolas Vasilache, Jeff Johnson, Michaël Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. 2015. Fast Convolutional Nets With fbfft: A GPU Performance Evaluation. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.7580

[21] Yulin Zhang and Xiaoming Li. 2020. Fast Convolutional Neural Networks with Fine-Grained FFTs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)* (Virtual Event, GA, USA) *(PACT '20)*. Association for Computing Machinery, 255–265. https://doi.org/10.1145/3410463.3414642

Received 2024-09-11; accepted 2024-11-04