Article

Efficient Convolutional Neural Networks Utilizing Fine-Grained Fast Fourier Transforms †

Yulin Zhang [1,2], Feipeng Li [1,2], Haoke Xu [3], Xiaoming Li [3] and Shan Jiang [1,2,*]

1 Key Laboratory of Ethnic Language Intelligent Analysis and Security Governance, Ministry of Education, Minzu University of China, Beijing 100081, China; yzhang@muc.edu.cn (Y.Z.)
2 School of Information Engneering, Minzu University of China, Beijing 100081, China
3 Electrical & Computer Engineering, University of Delaware, Newark, DE 19716, USA
* Correspondence: jshan@muc.edu.cn

† This paper is an extended version of our paper published in Zhang, Y.; Li, X. Fast Convolutional Neural Networks with Fine-Grained FFTs. In Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT '20), Virtual Event, GA, USA, 3–7 October 2020; pp. 255–265.

Abstract: Convolutional Neural Networks (CNNs) are among the most prevalent deep learning techniques employed across various domains. The computational complexity of CNNs is largely attributed to the convolution operations. These operations are computationally demanding and significantly impact overall model performance. Traditional CNN implementations convert convolutions into matrix operations via the im2col (image to column) technique, facilitating parallelization through advanced BLAS libraries. This study identifies and investigates a significant yet intricate pattern of data redundancy within the matrix-based representation of convolutions, a pattern that, while complex, presents opportunities for optimization. Through meticulous analysis of the redundancy inherent in the im2col approach, this paper introduces a mathematically succinct matrix representation for convolution, leading to the development of an optimized FFT-based convolution with finer FFT granularity. Benchmarking demonstrates that our approach achieves an average speedup of 14 times and a maximum speedup of 17 times compared to the regular FFT convolution. Similarly, it outperforms the Im2col+GEMM approach from NVIDIA's cuDNN library, achieving an average speedup of three times and a maximum speedup of five times. Our FineGrained FFT convolution approach, when integrated into Caffe, a widely used deep learning framework, leads to significant performance gains. Evaluations using synthetic CNNs designed for real-world applications show an average speedup of 1.67 times. Furthermore, a modified VGG network variant achieves a speedup of 1.25 times.

Keywords: GPU; fast Fourier transform; convolutional neural network; algorithm optimization

check for updates

Citation: Zhang, Y.; Li, F.; Xu, H.; Li, X.; Jiang, S. Efficient Convolutional Neural Networks Utilizing Fine-Grained Fast Fourier Transforms. *Electronics* **2024**, *13*, 3765. https://doi.org/10.3390/electronics13183765

Academic Editor: Maciej Ławryńczuk

Received: 6 August 2024
Revised: 9 September 2024
Accepted: 19 September 2024
Published: 22 September 2024

1. Introduction

In recent years, deep convolutional neural networks (CNNs) have emerged as a powerful tool within deep learning, significantly impacting computer vision applications. CNNs excel in image processing and pattern recognition tasks, revolutionizing fields from image classification to object detection and semantic segmentation. For instance, in image classification, models like EfficientNet [1] and ResNet [2] achieve top performance on benchmarks like ImageNet [3]. In object detection and localization, architectures such as Faster R-CNN [4], YOLO (You Only Look Once) [5], and EfficientDet [6] accurately identify objects and their boundaries across various scales and contexts. Semantic segmentation tasks extensively utilize CNNs for pixel-level classification, with notable architectures like U-Net [7] and DeepLab [8], incorporating with attention mechanisms and advanced feature aggregation techniques to achieve state-of-the-art results in segmenting complex scenes. CNNs have also made significant strides in generating realistic images and enhancing image quality. Generative Adversarial Networks (GANs) like StyleGAN [9] and BigGAN [10] utilize CNN architectures to generate high-resolution images with detailed and coherent structures. In autonomous vehicles [11,12], CNNs are crucial for perception tasks such as lane detection, object recognition, and path planning, designed to operate efficiently in real-time scenarios. Figure 1 illustrates the diverse applications where CNNs have made significant advancements. However, as CNNs increase in complexity, their computational demands and the associated memory requirements increase exponentially. This can pose challenges when deploying the model on embedded devices with limited available memory. Furthermore, in the context of real-time interactive applications, it demands a smooth and seamless user experience, where interactions and responses occur almost instantaneously. In such applications, long latency can significantly disrupt the user experience, leading to frustration and disengagement. Therefore, optimizing the computational performance of CNNs is critical for both their development in research and their practical deployment across various applications.

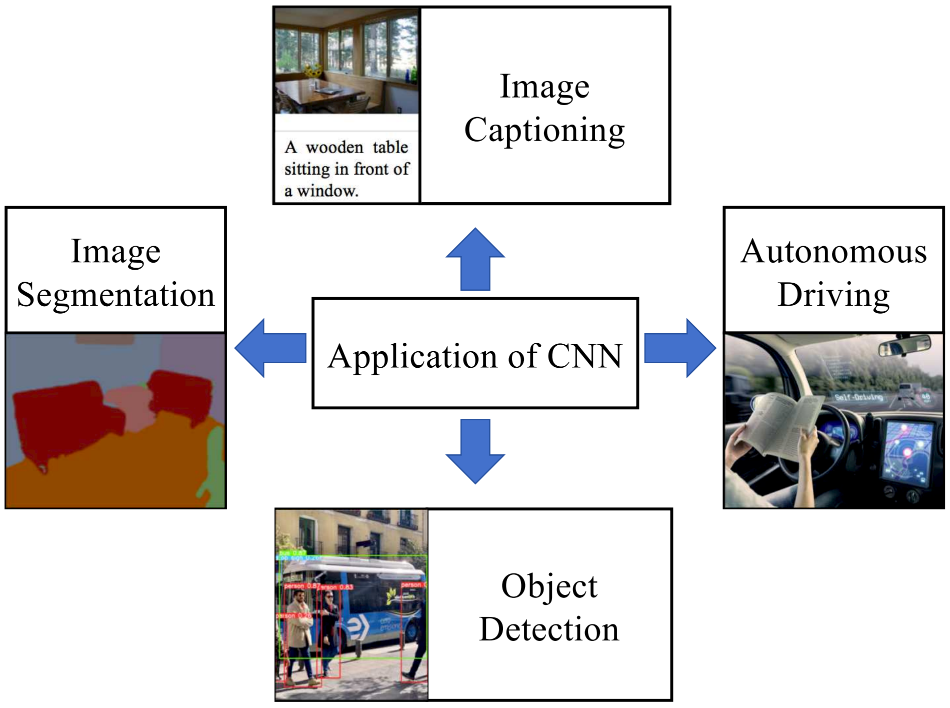


Figure 1. CNNs have extensive applications across diverse real-world scenarios, including image captioning, autonomous driving, object detection, and image segmentation.

Convolutional neural networks (CNNs) comprise a series of stacked layers, each playing a specific role in the learning process. However, convolutional (CONV) layers [13–19] consume a significant portion of the computation. Despite its relatively low parameter count, convolution is the most time-consuming operation in convolutional neural networks, significantly impacting the overall computational workload during network training and inference. Several studies, including [17,18], have empirically demonstrated that convolutional layers are the computational bottleneck in CNNs. These layers can consume about 90% of the execution time during both the forward and backward passes. This substantial computational cost becomes even more pronounced with the growing depth and complexity of modern CNNs. Consequently, prior research has extensively focused on developing techniques to optimize the convolution process.

Considering the computational demands of CNNs, a common solution leverages the parallel processing capabilities of Graphics Processing Units (GPUs). GPU acceleration has significantly impacted the performance of CNNs, enabling faster computations, larger models, and real-time applications. Among various implementations of CNNs on GPUs, NVIDIA's cuDNN library [20] provides a highly compatible solution for deep learning acceleration on their hardware. Since most deep learning frameworks integrate GPU support by default [21–26], achieving high-performance convolution on GPUs becomes a critical factor for maximizing overall CNN performance. On the other hand, to reduce computational complexity and enhance the speed of convolutional layers, various studies have concentrated on employing approximation algorithms or quantization techniques, though these methods may lead to some accuracy degradation. They are considered orthogonal and complementary to our direction of optimizing convolution algorithms. For instance, Sabir et al. [27] introduce TiQSA, a method that combines tile quantization and symmetry approximation with a Particle of Swarm Convolution Layer Optimization algorithm to significantly reduce computational workload without substantial accuracy loss. Additionally, Gysel et al. [28] propose Ristretto, a framework for model approximation that quantizes CNN layers into fixed point arithmetic to compress models and reduce execution time. Furthermore, Limonova et al. [29] and Cintra et al. [30] introduce methods to transform convolutional network structures and approximate convolutional neural networks for reduced computational complexity while maintaining accuracy.

Optimizing the fundamental execution of convolution constitutes another area of research, highlighted by two key methods: (1) Utilizing FFT for performing convolutions in the Fourier domain [31–33], with studies [16] demonstrating that larger kernels can lead to greater improvements. However, as CNNs often utilize smaller kernels [34,35], the Winograd algorithm has been introduced to decrease multiplications at the expense of more additions. This trade-off may not provide sufficient benefits for larger kernels. (2) Direct calculation of convolution through matrix–matrix multiplication is another method [32], converting convolution into matrix multiplication to leverage the capabilities of optimized GEMM libraries for efficient computation. In addition to these main methods, another effective approach is auto-tuning [36]. This technique automatically selects the optimal convolution algorithm based on real-time performance measurements on the actual hardware. Given that each convolution algorithm is best suited to specific scenarios, and the optimal choice can vary depending on the particular layer parameters and hardware capabilities, auto-tuning offers a tailored optimization strategy that enhances computational efficiency. This study focuses on investigating performance optimization for the convolution process within CNNs on the backend implementation level. A critical observation in this work is the presence of significant yet intricately patterned redundancies within the matrix representation used in CNN convolutions. This pattern, previously overlooked in the field, presents an opportunity for computational optimization. A thorough examination of this redundancy uncovers a doubly block Hankel matrix data structure. By leveraging this newly identified matrix structure, we propose a novel FFT-based convolution algorithm that utilizes this data pattern. Unlike regular approaches that utilize a 2D FFT across the entire feature map, our algorithm leverages fine-grained FFTs. Compared with existing state-of-the-art implementations, our fine-grained FFT approach delivers substantial performance enhancements. Extensive comparisons confirm the advantage of our fine-grained FFT approach, demonstrating its outperformance on both synthetic and real-world benchmarks.

From the point of view of accelerating the performance of CNNs, this work offers the following key contributions:

(1) We introduce an optimization of the CNN's convolution process by identifying and leveraging hidden redundancy within unrolled feature maps and present a new FFT-based convolution method that reduces computational complexity and improves CNN performance. Our method achieves a maximum speedup of up to 17 times compared to cuDNN convolution methods.

(2) A salient aspect of our proposed algorithm is the ability to operate directly on the input images without unfolding the images into column vectors, namely im2col. We can avoid the significant redundancy that arises from im2col transformations. Our approach enables more efficient data reuse and minimizes memory usage. Specifically, the memory savings scale asymptotically with the kernel size.

(3) In order to further optimize the performance, we implement an auto-tuning framework to automatically tune the CUDA parameters of the implementation. By intelligently searching the design space, we are able to find a configuration that yields additional improvements beyond the base fine-grained FFT implementation.

(4) The proposed method can be easily applied in a deep learning framework by replacing the current convolutions with our method.

On the other hand, the work in this paper is an extension of our previous publication [37]. Here, compared with our previous work, we also present new contributions that extend the understanding and applicability of our method:

(1) In-depth method description: Due to space limitations in the conference paper format, the original method description was concise. In this work, we address this gap by expanding every section from the conference version, providing a comprehensive and detailed explanation of our method.

(2) Broader performance evaluation: We expand the evaluation by including experiments on additional GPUs. This broader range of testing strengthens the algorithm's high performance and generalizability across different hardware platforms.

2. Overview and Background

In this part, we begin by delving into the core concept of convolution and then provide an overview of convolutional neural networks (CNNs), followed by a detailed examination of various convolution algorithms. To ensure clarity throughout the paper, a summary of notations used in this study is provided in Table 1.

Table 1. Summary of notations used in this work.

Name	Description
C	Input channels
H	Input height
K	Number of kernels
N	Mini-batch size
P	Padding
Q	Output width
R	Output height
S	Stride
U	Kernel height
V	Kernel width
W	Input width

2.1. Convolution

In the context of convolutional neural networks (CNNs), convolution refers to the fundamental operation of applying a filter (also known as a kernel) to input data. This process involves element-wise multiplication of the filter with the corresponding receptive field in the input, followed by summation to generate a feature map. This localized computation captures spatial relationships between pixels, enabling the network to capture features such as edges, textures, or more complex structures within the data. Through this process, it progressively assembles them into higher-level representations. Convolution operations play a crucial role in feature detection within CNNs, enabling the network to acquire hierarchical representations of input data, ultimately leading to the extraction of essential higher-level features for accurate classification and prediction tasks in the fields of machine learning and computer vision.

A two-dimensional (2D) convolution is the core operation in CNNs, which is a mathematical operation that extracts local features and patterns from input data. It involves performing a dot product between the filter and the corresponding values in the feature map. This operation systematically moves the kernel across the input image in both horizontal and vertical directions. As the kernel slides, the 2D convolution operation is applied at each position. Initially, the kernel is placed over the top-left corner of the input image. It moves one step at a time across the image horizontally until it reaches the rightmost position, and then the kernel resets to the leftmost position of the next row down. The process is repeated, and the kernel continues moving from left to right and then from top to bottom until the bottom right corner is covered. The convolution process within CNNs fundamentally consists of batched 2D convolutions. This involves executing a 2D convolution for each input channel against the filter, subsequently aggregating the outcomes from all channels. When multiple filters are used, the resulting output is formed by concatenating the individual output matrices generated by convolving the input with each corresponding filter.

2.2. Convolutional Neural Network

In a conventional CNN architecture, each stage consists of a convolutional layer followed by a non-linearity layer and a pooling layer. This structure enhances the robustness and invariance of the learned features to small shifts and distortions within the data. Convolutional layers, equipped with various kernels, act as feature extractors, identifying specific patterns within the input. Each kernel in a convolutional layer targets a specific feature, allowing multiple kernels to extract a diverse set of feature maps. These feature maps progress from representing low-level details in the early layers to capturing more abstract and complex features in the higher layers. The inputs and outputs of these layers are known as feature maps, where each neuron connects only to a localized region of the previous layer, known as the local receptive field. This sparse connectivity [38] significantly reduces the number of connections needed between layers. Another defining characteristic of CNNs is parameter sharing [38], where a single set of weights is applied across all spatial positions of the input feature maps to generate the output feature maps. This approach not only accelerates computations but also minimizes the memory footprint of the network's parameters. The number of output feature maps in a convolutional layer directly corresponds to the number of kernels used, with each kernel generating a specific extracted feature. In the final stage, a fully-connected layer integrates the high-level features extracted by the convolutional layers to perform the classification task.

2.3. Direct Convolution

Direct convolution, as the name suggests, involves the straightforward application of the convolution operation between the input data and kernels. It involves the explicit computation of the dot product between the elements of the kernels and the input data. It can be computationally expensive, especially for large input sizes and many filters, due to the high number of multiplications and additions required.

A naive implementation of convolution can be implemented using seven nested for loops, as illustrated in Algorithm 1. The outer loops traverse the input image, while the inner loops traverse the kernel and perform accumulation. This direct method of convolution does not require extra memory overhead; however, it suffers from low performance due to non-sequential data access. Executing direct convolution on GPUs essentially translates to parallelizing the execution of for-loops. The loops outlined in lines 1–4 of Algorithm 1 are independent, making them highly suitable for parallel execution on GPUs. However, lines 5–7 are not independent and present a shared data dependency. Due to loop-carried dependencies across iterations, sufficient parallelism is not exposed, making it challenging to fully utilize GPU resources when implementing the convolution on GPUs. Cuda-convnet [39] stands as an early example of a CNN framework utilizing direct convolution. It demonstrates high efficiency with larger batch sizes [20], yet this efficiency diminishes for batch sizes of 64 or below. Georganas et al. [40] address this limitation by exploring optimization techniques for direct convolution on SIMD architectures. They focus on improving computational efficiency and reducing latency in convolution execution. In contrast, Lavin et al. [41] take a different approach, developing maxDNN, an efficient convolution leveraging SGEMM implementations from an open-source assembler designed for NVIDIA Maxwell GPUs [42]. By exploiting hardware-specific optimizations available within the Maxwell architecture, maxDNN achieves high-performance gains. However, this optimization comes at the cost of portability, as it is not applicable to other architectures.

Algorithm 1 Direct convolution by seven nested for loops (see Table 1 for notations)

```
1: for $n = 0$ to $N - 1$ do
2:   for $k = 0$ to $K - 1$ do
3:     for $c = 0$ to $C - 1$ do
4:       for $r = 0$ to $R - 1$ do
5:         for $q = 0$ to $Q - 1$ do
6:           for $u = 0$ to $U - 1$ do
7:             for $v = 0$ to $V - 1$ do
8:               $Output[n, k, r, q] \mathrel{+}= Kernel[k, c, u, v] \cdot Input[n, c, r + u, q + v]$
```

2.4. Im2col+MM Convolution

Im2col+GEMM is a widely used technique for performing convolution efficiently. First, image patches are extracted based on the kernel size and then reshaped into column vectors. These column vectors are subsequently concatenated to form a single matrix. This process, commonly referred to as lowering or unrolling, effectively transforms the convolution operation into a general matrix multiplication (GEMM). Im2col unrolls the input images into 2D matrices, while the kernel remains in its original matrix form (kernel matrix). As a result, the convolution process is simplified to a single GEMM operation, with each row in the resulting output matrix corresponding to a unique output feature map. The width of the output feature map can be determined using the following formula.

$$Q = (W - V + 2P)/S + 1 \tag{1}$$

where Q, W, V, P, and S are defined in Table 1. The last set of three hyper-parameters determines the width of output feature maps. While our focus here is on width, the concept can be easily extended to height as well. Figure 2 demonstrates the im2col+MM convolution method. The kernel matrix contains four different kernels, with each row representing a distinct kernel. The unrolled input matrix is created through the im2col process. Multiplying these two matrices results in each row of the output matrix representing an output feature map corresponding to a specific kernel. This method leverages highly optimized general matrix multiplication routines, which are well-supported and optimized in most hardware architectures, offering a significant speedup in computation. Moreover, it leverages highly optimized linear algebra libraries. However, it is worth noting that this method can increase memory usage due to the duplication of data in the im2col step. Despite increasing the memory footprint, the gains in computational efficiency make this approach highly attractive for implementing CNNs.

By multiplying the unrolled input with the kernel matrix, each row in the resulting output matrix corresponds to the convolution of the input with its respective kernel. CNNs use kernels of size K×CUV to analyze data, and the input data (NCHW format) is flattened into N slices, each with size CUV×NRQ (see Table 1 for notations). It is important to note that this process can introduce redundancy, with each element potentially being replicated up to UV times. As a result, larger kernel sizes cause more duplication and increase the temporary memory needed to store the unrolled input data. The core idea behind Im2col+MM lies in transforming the convolution operation into a well-established matrix multiplication by unrolling and duplicating the input. Thus, this method is largely dependent on the efficiency of the underlying GEMM implementation. However, the performance of the cuBLAS matrix multiplication routine does not always increase linearly with slight variations in matrix dimensions [43]. This is because cuBLAS selects an optimized implementation from a pre-defined set based on the specific input matrix dimensions. In some cases, the unrolled matrix dimensions generated by Im2col might not align perfectly with these optimized implementations, potentially leading to sub-optimal performance in cuBLAS.

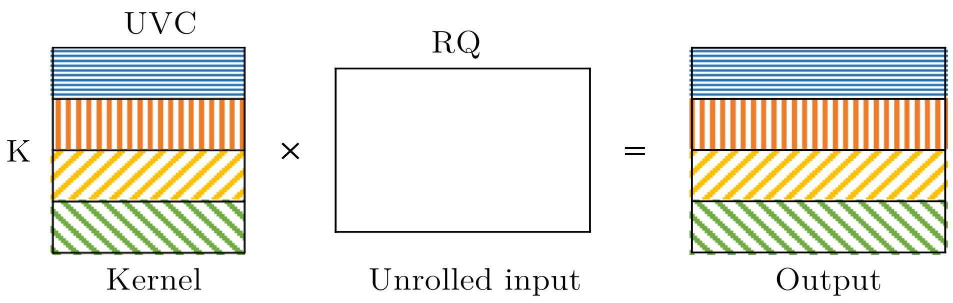


Figure 2. The kernel matrix has four different patterns with distinct colors, each row representing one kernel with dimension UVC. By multiplying the unrolled matrix, each row in the output matrix represents an output feature map. Refer to Table 1 for notations.

The Im2col technique, which transforms convolution operations into matrix multiplications, has been shown to improve CNN performance. This concept was first identified in [32] and independently discovered by Yanqing et al. [21] in their work on the Caffe deep learning framework. A significant advantage of this approach is its ability to exploit highly optimized linear algebra libraries like cuBLAS, leading to performance improvements. However, it is important to consider the resource consumption associated with im2col. The image-to-column transformation process introduces a substantial memory overhead, which can be a limiting factor in certain scenarios. Vasudevan et al. [44] discuss an alternative to im2col that avoids its high memory footprint by applying convolution kernels directly to input images, demonstrating the inefficiency in memory usage of the im2col approach. Similarly, Wang et al. [45] present a parallel convolution algorithm that aims to reduce memory footprints and improve packing efficiency compared to im2col+GEMM methods. Zhao et al. [46] propose a method saving over 60% of hardware storage space compared to im2col, highlighting the method's inefficiency in large-scale convolutions.

2.5. FFT Convolution

FFT-based convolution [16,31] leverages the mathematical properties of the Fourier transform to transform the convolution operation in the spatial domain into a simpler element-wise multiplication in the frequency domain, thus significantly reducing the number of operations required for convolution and accelerating the processing time. It utilizes the Fast Fourier Transform (FFT) to efficiently compute element-wise multiplications in the frequency domain, which, according to the convolution theorem, are equivalent to spatial convolutions.

$$\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g) \tag{2}$$

Here, $\mathcal{F}$ denotes the Fourier transform, $*$ denotes the convolution operation, and $\cdot$ denotes element-wise multiplication. Inverse Fourier transforming the product of the Fourier transforms of f and g yields the convolution of f and g in the spatial domain. A key aspect of this approach is the prerequisite of equal-sized input and weight tensors. To satisfy this requirement, zero padding is employed on the tensors before transformation. This method involves converting both the input and weight tensors from the spatial domain to the frequency domain using the Fast Fourier Transform (FFT). Once transformed, a pointwise multiplication is performed, which involves multiplying the corresponding elements of the transformed input with the complex conjugate of the transformed kernel. The final step involves applying the inverse FFT, which transforms the data back into the spatial domain.

The FFT-based approach significantly decreases the algorithmic complexity of performing convolution in the spatial domain. It reduces the computational complexity of convolutions from $O(n^2)$ in the spatial domain to $O(nlogn)$ in the frequency domain, where n is the size of the input. However, a significant drawback of this approach is its high memory footprint. This primarily arises due to the requirement for the input and weight tensors to have identical sizes. The process involves padding the weight tensor to match the dimensions of the input tensor. When the input tensor is considerably larger than the weight tensor, this padding can lead to substantial memory overhead. This excessive padding can also negatively impact the efficiency of FFT-based convolution. Moreover, the overhead associated with performing FFT and inverse FFT (IFFT) can outweigh the computational savings. To overcome this limitation, a tiling strategy [47] is often implemented. Tiling essentially decomposes the large convolution into a sequence of smaller convolutions. By processing these smaller convolutions individually, the memory footprint is reduced, enabling more efficient computations. Additionally, extra memory is needed to store the FFT coefficients, which can significantly increase the memory footprint, especially for high-dimensional data. In our work, we leverage the inherent symmetry of real inputs within the Fourier domain to decrease the required storage for FFT coefficients by half. Additionally, this symmetry is used to lower the computational expense of the pointwise multiplication.

FFT convolution remains a compelling approach, particularly for applications where computational resources are a bottleneck. Abtahi et al. [48] evaluate FFT-based convolution against direct and overlap-add methods, demonstrating substantial performance benefits on embedded platforms. FFT-based convolution methods apply the Fourier transform to accelerate the convolution operation itself but may not maintain the entire network's operations within the Fourier domain. In contrast, the Fourier Convolutional Neural Networks (FCNNs) [49] framework is designed to operate entirely within the Fourier domain, meaning that both the forward and backward passes of the network are conducted in this domain. The proposed FCNNs significantly speed up the training process without sacrificing effectiveness.

2.6. Winograd Convolution

Winograd Convolution [33], based on the Winograd minimal filtering algorithm [50], is designed to minimize the arithmetic complexity of convolutions by minimizing the number of multiplicative operations. It starts by applying linear transformations to the input feature map and the convolutional kernel. After both the input and the kernel have been transformed, the algorithm proceeds with an element-wise multiplication of the transformed representations. Unlike traditional convolution, which involves numerous multiplicative operations, the Winograd algorithm substantially reduces the number of these operations. The last step converts the data back from the Winograd-specific representation to the original spatial domain. The Winograd algorithm's efficiency stems from the fact that the computational savings on multiplications outweigh the overhead introduced by the additional additions and transformations. However, reshaping both the kernel and image patch into matrices requires additional memory. Pre-computed constants further increase the memory footprint. This can be a significant concern for large CNNs or resource-constrained environments. Additionally, the matrix multiplication approach in Winograd can introduce rounding errors compared to the standard convolution.

3. Motivation

In this section, we explore how data redundancy is identified in the im2col-based convolution and how this new insight inspires a more efficient approach to performing convolutions. Recall that the im2col operation restructures the input feature map into a sequence of columns, each representing local patches of the input, while the kernel data are already formatted into a matrix. It transforms convolution operations into matrix multiplications to leverage the power of optimized GEMM libraries. During im2col, due to the overlapping of receptive fields, certain elements are repeated across multiple columns, creating redundancy. Additionally, the im2col process sometimes replicates zeros when the input feature map is padded with zeros to maintain spatial dimensions and preserve border information. By understanding the distribution of zeros, we can avoid unnecessary multiplications that yield zero. For instance, Figure 3 shows how a 3×3 input, with a zero padding of 1, is expanded into a large output featuring redundancy. This expansion is achieved using the im2col process with a 2×2 kernel. However, the specific amount of redundancy introduced by this im2col process is unknown.

0	0	0	0	0
0	1	2	3	0
0	4	5	6	0
0	7	8	9	0
0	0	0	0	0

$\xrightarrow{\text{im2col}}$

0	0	0	0	0	1	2	3	0	4	5	6	0	7	8	9
0	0	0	0	1	2	3	0	4	5	6	0	7	8	9	0
0	1	2	3	0	4	5	6	0	7	8	9	0	0	0	0
1	2	3	0	4	5	6	0	7	8	9	0	0	0	0	0

Figure 3. A 3×3 input matrix is padded with zeros of size one (shown in blue). The matrix on the right is generated using a 2×2 kernel through the im2col process and exhibits redundancy, the details of which are unspecified.

To analyze and leverage the redundancy to develop a more efficient convolution strategy, several fundamental questions naturally arise: (1) Which elements in the output matrix are essential, and which are duplicates? (2) How can we mathematically describe the redundancy in the output matrix? In the following sections, we introduce a recursive data pattern to characterize the observed redundancy. This pattern allows us to design a new convolution algorithm that avoids unnecessary computations and data storage. This approach ultimately leads to our "fine-grained FFT convolution".

4. A New Data Pattern

This section first analyzes how the im2col process works in detail and then explores two main types of redundancy that occur during this process caused by the way im2col iterates over the feature map with the kernel. After revealing the zero distribution within the output matrix, we will introduce a concise mathematical framework to describe these redundancies. This framework reveals a connection between the redundancy and a specific type of matrix structure—the doubly block Hankel matrix [51]. These insights establish a theoretical foundation to optimize convolutional operations in CNNs.

4.1. Im2col Process

Before understanding the redundancy and subsequently diving into the specifics of the new data pattern, let us revisit the im2col operation, as it serves as the foundation for this pattern. The im2col operation stands for "image to column", and it stretches the local receptive field or patch in the original image into column vectors. Specifically, im2col takes each local patch, defined by the size of the convolutional kernel, and stretches it out into a column. Consequently, an input image is transformed into a matrix where each column corresponds to one local receptive field or patch. As shown in Figure 4, the im2col process transforms a patch containing elements numbered from one to nine into a column vector. This involves transposing and concatenating each row, distinguished by different colors, into a single extended column. Each row from the patch is sequentially transposed and aligned to create a continuous vertical array.

In the im2col process, the kernel slides across the feature map both horizontally and vertically, allowing each row of the kernel to function independently. This means that a two-dimensional kernel can be conceptualized as a collection of separate one-dimensional row kernels. As the kernel traverses the feature map, each individual row kernel interacts with corresponding sections of the feature map to compute outputs. The final output generated by im2col is a combination of the results from each of these row kernels. This process creates a specific type of redundancy within each row, referred to as intra-row redundancy.

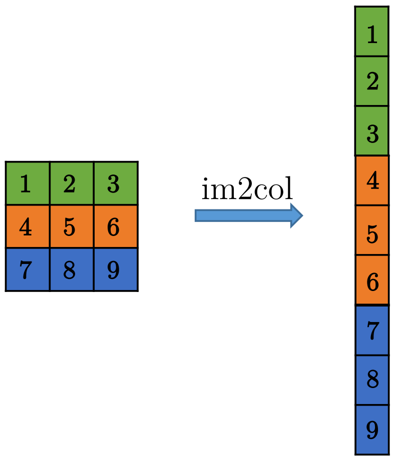


Figure 4. The im2col process reshapes a 3×3 input patch into a column vector by stacking each row column-wise into a single column vector.

4.2. Intra-Row Redundancy

Figure 5 demonstrates the occurrence of intra-row redundancy as a row kernel traverses a single-row feature map. Let us consider that the feature map has a length m and the kernel has a length n, with a stride of 1. As the kernel slides across the feature map from one position to the next, specifically from position N (indicated with a black rectangle) to $N + 1$ (indicated with a red rectangle), there are $n - 1$ elements that overlap between the two positions, which are highlighted by a green stripe. This overlap results in only the leftmost element of position N and the rightmost element of position $N + 1$ being different. As the kernel moves horizontally, the im2col operation systematically rearranges these elements into columns. This rearrangement places the elements side by side, resulting in each new column sharing $n - 1$ overlapping elements with its predecessor, but each overlapped element is shifted upwards by one position. As a result, the elements along the skew diagonals remain constant in the generated matrix.

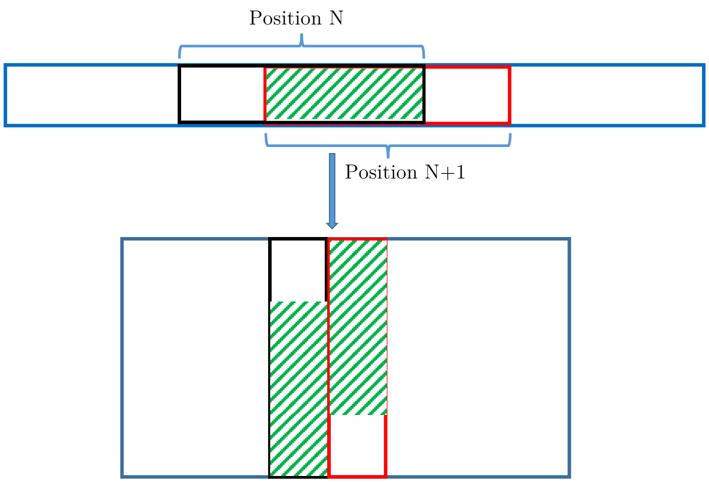


Figure 5. The schematic representation of how intra-row redundancy occurs. In the upper diagram, the black rectangle symbolizes the row kernel at position N, while the red rectangle represents the row kernel at position $N + 1$. The green stripe highlights the overlapping elements between these two positions. In the lower part, the green stripe illustrates the identical elements found in adjacent columns of the resulting matrix.

Let us mathematically express the intra-row redundancy discussed above. The size of the row kernel is V. It is used to generate the unroll matrix G with dimensions of V by Q. Considering the elements of G, we can denote the element at row i and column j as $G_{i,j}$. Here, we focus on the case where i $\leq$ j. In this scenario, we observe that $G_{i,j} = G_{i+k,j-k}$ for all valid values of $k = 0, ..., j-i$.

4.3. Inter-Row Redundancy

During the convolution operation, the kernel starts at the top-left corner of the feature map and moves across the feature map from left to right. Throughout this movement, each row of the kernel experiences a pattern of redundancy, as depicted in Figure 5. Upon reaching the feature map's right edge, the kernel moves downward by one row, where a different form of redundancy occurs due to the overlapping traversal of elements by successive row kernels, which we term inter-row redundancy. This pattern continues until the kernel reaches the bottom right corner of the feature map. In the example shown in Figure 6, the kernel consists of four differently colored rows, and the feature map is zero-padded with a width of one. As the kernel moves across and down the feature map, each row of the kernel (k_1, k_2, k_3 and k_4) contributes to generating separate output rows, corresponding to the first through the fourth rows in the resulting output matrix on the right in Figure 6. Within this output matrix, the blocks on the skew diagonal are identical, while each block displays intra-redundancy. Since the padding consists of zeros with a width of one, the top left and bottom right blocks of the matrix are zero matrices.

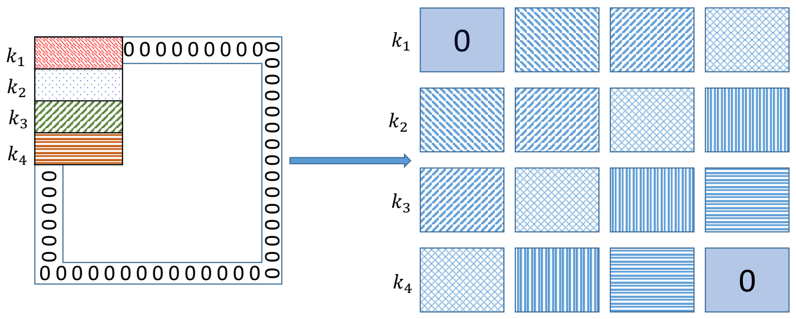


Figure 6. The schematic representation of inter-row redundancy. The kernel comprises four rows labeled k_1 to k_4. The feature map is zero-padded with a size of one on all sides. This padding results in the top-left and bottom-right blocks of the matrix being zero matrices, as illustrated. The blocks along the skew diagonal are identical, highlighted by the same pattern to denote inter-row redundancy. It also exhibits intra-row redundancy within each block.

Next, we mathematically summarize the inter-row redundancy within the generated matrix G, which has dimensions $UV \times QR$. Let $G_{i,j}$ represent the block located at the i-th row and j-th column within G. We focus on the case where i $\leq$ j. Under these conditions, it is observed that $G_{i,j}$ is equal to $G_{i,j} = G_{i+k,j-k}$ for all k ranges from 0 to $j-i$. Each individual block within G also exhibits intra-row redundancy, as discussed in Section 4.2. This inter-row redundancy builds upon the intra-row redundancy within each block.

4.4. Im2col-Based Convolution Redundancy

In convolutional neural networks (CNNs), the convolutional layers perform batched 2D convolutions. In this process, each input feature map is convolved with the corresponding channel of the kernel, and the results from these batched 2D convolutions are aggregated to form the output feature map. To facilitate this, the im2col method is utilized, which replicates each feature map uniformly and generates expanded matrices. As illustrated in Figure 7 on the right, these matrices exhibit both intra-row and inter-row redundancy. This redundancy follows a consistent data pattern, characterized by identical elements or blocks along the skew diagonals.

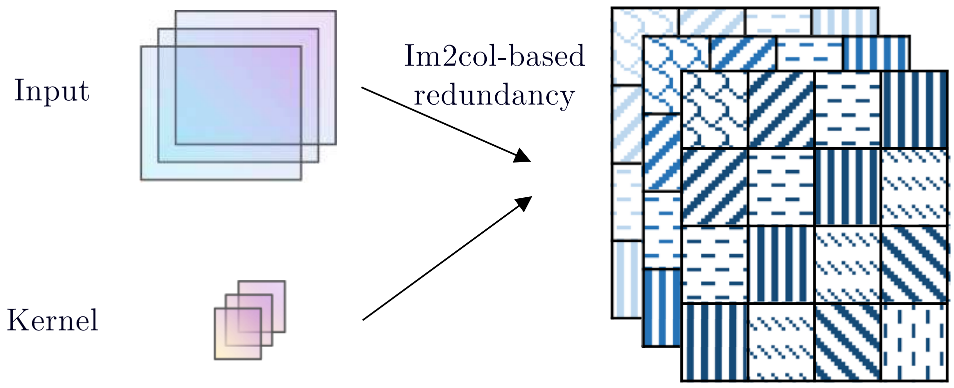


Figure 7. Illustration of the im2col process for a three-channel input and kernel. It emphasizes the resulting redundancies in the unrolled output matrix. Each channel of the output shows inter-row redundancy, characterized by identical blocks along skew diagonals. Furthermore, each block within the output matrix exhibits intra-row redundancy. The combination of inter-row and intra-row redundancies is collectively referred to as Im2col-based convolution redundancy.

In this section, we adopt a systematic bottom-up approach to explore the redundancy inherent in the im2col operation as it is applied within CNNs. Our analysis begins by examining the basic 1D convolution, where we identify patterns of intra-row redundancy. We then progress to analyzing 2D convolutions, revealing the presence of inter-row redundancy. Since CNNs typically employ batched 2D convolutions, our examination extends to these, where we observe im2col-based convolution redundancy, which effectively integrates both inter-row and intra-row redundancies. While redundancy motivates us to design novel algorithms to minimize memory usage, its greater significance lies in providing an avenue for optimizing convolutions within CNNs. By meticulously identifying and addressing these redundancy patterns, we enable the design of more efficient computational strategies.

4.5. Doubly Block Hankel Matrices

Now that we have identified repetitive patterns within the data, specifically intra-redundancy and inter-redundancy. We are equipped to address the questions from Section 3 about how these redundancy patterns are represented and how we can use them. The redundancy pattern, characterized by both intra-row and inter-row redundancies, can be qualitatively described as follows: Each block along the skewed diagonals in Figure 8 is identical. Within these blocks, there is intra-row redundancy, with elements along these diagonals remaining constant. Building on this qualitative description, our next step involves a quantitative analysis to define the structure of these redundancy patterns.

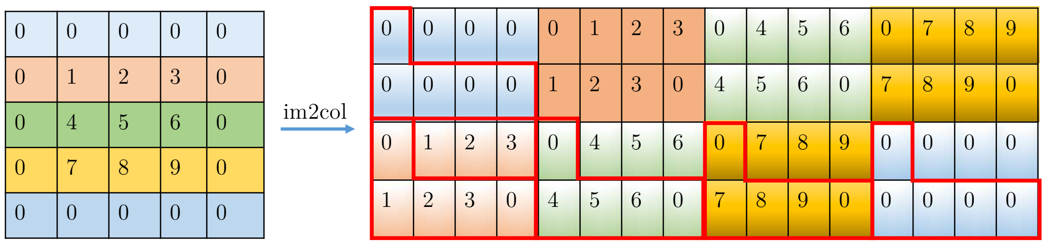


Figure 8. On the left side of the figure is the original feature map, and on the right side is the unrolled matrix generated by the im2col operation. Each row within the original feature map is highlighted with a specific color, and elements in these rows correspond to blocks of the same color in the unrolled matrix. Each block of the same color is identical, and elements along the skew diagonals within these blocks remain constant. Notably, the elements bounded by the red lines on the right correspond to each row in the original feature map.

In the im2col operation, the output matrix generated by the process displays a pattern of inter-row redundancy. Additionally, each individual block within this matrix exhibits intra-row redundancy. The zero elements, which are added to handle the boundaries of the input feature map, are primarily located in the top-left and bottom-right corners of the matrix. The zero elements also appear in the corresponding corners within each non-zero block. With these insights, we can track every element of the input data more effectively. It allows for a comprehensive analysis of the data structure created by the im2col process, facilitating convolution operations in neural networks. To define this process more formally, let us consider a feature map of size $m \times m$ and a kernel of size $n \times n$. The feature map is zero-padded with size p. The im2col operation then reshapes the feature map into a matrix with dimensions $n^2 \times (m - n + 2p + 1)^2$, where each individual block in the resulting matrix has dimensions of $n \times (m - n + 2p - 1)$.

To formalize the im2col operation process, let us consider a feature map of size $m \times m$ and a kernel of size $n \times n$. The feature map is zero-padded with size p, and the stride for the convolution is set to one. The im2col operation then reshapes this padded feature map into a matrix with dimensions $n^2 \times (m - n + 2p + 1)^2$, where each individual block in the resulting matrix has dimensions of $n \times (m - n + 2p + 1)$. The mathematical definitions for intra-row and inter-row redundancy can be described as follows:

$$Output_{intra|inter}[i][j] = Output_{intra|inter}[i-1][j+1] \tag{3}$$

$i > 0, j > 0$ and $i < n, j < m - n + 2p - 1$. The padded zero distribution follows the equation.

$$Output_{intra|inter}[i][j] = 0 \tag{4}$$

such that $i < p, j < p$ or $i > n - p - 1, j > m - n + p$. i and j denote the indices for the elements and blocks in intra-row and inter-row redundancy, respectively.

We have made a significant discovery in our analysis of the im2col operation. The patterns we observed, captured by Equation (3), demonstrate intra-row and inter-row redundancy and align with the definition of a Hankel matrix [51]. In a Hankel matrix, each skew-diagonal contains constant values. This characteristic implies that for any given skew-diagonal, all elements in that diagonal are identical. When we apply the im2col operation to an input feature map, the resulting output matrix is not just any matrix, but specifically a doubly block Hankel matrix. This means the matrix exhibits a Hankel structure at two levels: (1) Individual block structure: each block within the matrix is a Hankel matrix. Within each block, the elements along any skew-diagonal are constant. (2) Overall matrix structure: the entire matrix, when considering all the blocks collectively, also adheres to the Hankel structure.

To illustrate this, an $n \times n$ Hankel matrix can be represented in the following form:

$$H = \begin{pmatrix} a & b & c & \cdots & g \\ b & c & d & \cdots & h \\ c & d & e & \cdots & i \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ g & h & i & \cdots & m \end{pmatrix} \tag{5}$$

In this matrix, each ascending skew-diagonal from the bottom-left to the top-right has the same value. For example, the first skew-diagonal from the bottom-left corner consists of the element g, the next skew-diagonal consists of h, and so on. The elements of an $n \times n$ Hankel matrix are determined by a sequence of length $2n - 1$. In the matrix H_{Block}, all H_{ij} are Hankel matrices. This means that each sub-matrix within H_{Block} maintains the Hankel property of constant skew-diagonals. This dual-level Hankel structure results in a doubly block Hankel matrix.

$$H_{Block} = \begin{pmatrix} H_{11} & H_{12} & \cdots & H_{1N} \\ H_{12} & H_{22} & \cdots & H_{2N} \\ \vdots & \vdots & \vdots & \vdots \\ H_{1N} & H_{2N} & \cdots & H_{NN} \end{pmatrix} \quad (6)$$

By exploiting this doubly block Hankel structure, the convolution operation is effectively translated into a multiplication between a Hankel matrix and a vector. This translation takes advantage of the inherent redundancy within Hankel matrices, allowing for a more computationally efficient multiplication compared to standard matrix-vector multiplication. To harness this efficiency, the Fast Fourier Transform (FFT) is utilized to significantly optimize the computation of Hankel-matrix-vector multiplication, resulting in substantial performance improvements for convolutions in CNNs. Detailed information about the proposed fine-grained FFT-based convolution method will be presented in the next section.

4.6. Data Correspondence

As discussed in Section 4.5, the unrolled matrix generated by the im2col process is indeed a doubly Hankel matrix. This indicates that each block within the matrix is a Hankel matrix in itself, and collectively, these blocks contribute to forming a larger Hankel matrix structure. Equally important, we identify the data correspondence between the original feature map and the expanded output. As illustrated in Figure 8, the elements bounded by the red lines on the right correspond to each row in the original feature map. This visualization demonstrates the data correspondence between the data before and after the im2col operation, highlighting how the im2col operation rearranges the elements in the feature map into the expanded output. Due to this clear data correspondence, it is not necessary to fully unroll the input feature map since all the elements can be retrieved from the input feature map. By exploiting this structure, it becomes possible to optimize memory usage by storing only unique elements or blocks, reducing redundancy, and avoiding the need to replicate data unnecessarily.

5. Fine-Grain-FFT-Based Convolution Algorithm

In this section, we delve into how the unique structure of Hankel matrices can be exploited to utilize the Fast Fourier Transform (FFT) for a more efficient matrix multiplication process. By recognizing the redundant data patterns within the Hankel matrices, we can transform the convolution into a more computationally efficient operation. We introduce an algorithm that leverages FFT to optimize the Hankel matrix-vector multiplication specifically for convolution operations in neural networks. This approach not only enhances computational efficiency but also significantly reduces memory overhead. Traditional methods require fully unrolling the input data, leading to substantial memory usage. Our fine-grained FFT-based convolution algorithm circumvents this by replacing the need for full data unrolling with an implicit element-wise matrix multiplication approach. Moreover, the theoretical analysis provided will demonstrate that our method offers a dual advantage: it reduces computational complexity by harnessing the power of FFT and minimizes memory overhead by eliminating the necessity of complete data unrolling. This results in a more efficient convolution operation, ultimately leading to performance improvements in convolutional neural networks (CNNs).

5.1. Hankel Matrix-Vector Multiplication

The core of our fine-grained-FFT convolution lies in the efficient multiplication of a kernel vector v with a Hankel matrix H. Hankel matrices, characterized by their structured format where each ascending skew-diagonal from left to right is constant, offer a more concise representation than general $n \times n$ matrices. To efficiently perform the multiplication involving a Hankel matrix, the Hankel matrix can be embedded into a larger circulant matrix. This transformation allows us to leverage the properties of circulant matrices to simplify the computation. Specifically, a circulant matrix is fully determined by its first row, referred to as the generating vector x. Each subsequent row in the circulant matrix is a cyclic right shift of the row above it, wrapping around at the end. One advantageous property of the circulant matrix C is that it is diagonalized by the Discrete Fourier Transform (DFT) matrix. The columns of the DFT matrix are eigenvectors for the circulant matrix, and the eigenvalues of the circulant matrix are obtained by taking the DFT of the generating vector x. The circulant matrix C is diagonalized by the Discrete Fourier Transform (DFT) matrix $\mathbb{F}$ [52], and it can be expressed as follows:

$$C = \mathbb{F}^{-1}\Delta\mathbb{F} \tag{7}$$

where Δ is a diagonal matrix containing the eigenvalues of C, such that $\Delta = \text{diag}(\mathbb{F}x)$. Therefore, the multiplication of the circulant matrix C with the kernel vector v is performed through a series of operations utilizing the Discrete Fourier Transform (DFT) and its inverse (IDFT). Specifically, it involves computing:

$$Xv = \mathbb{F}^{-1}(\mathbb{F}x \circ \mathbb{F}v) \tag{8}$$

The multiplication process can be summarized as follows: (1) DFT of the generating vector x. This step transforms x into the frequency domain, yielding the eigenvalues of the circulant matrix C. (2) DFT of the vector v. Apply the DFT to the vector v, resulting in $\mathbb{F}v$. (3) Element-wise multiplication. Perform the Hadamard (element-wise) multiplication, and this operation combines the transformed representations of x and v in the frequency domain. (4) Inverse DFT. Compute the inverse DFT of the product from the previous step to transform the result back into the spatial domain. This step yields the final result of the matrix-vector multiplication. These operations can be efficiently computed using the Fast Fourier Transform (FFT). It significantly reduces the computational complexity from $O(n^2)$ in the direct approach to $O(n \log n)$ due to the efficiency of the FFT.

5.2. Hankel Matrices to Circulant Matrices

As previously discussed, circulant matrix-vector multiplication can be efficiently computed using the Fast Fourier Transform (FFT). Given the computational advantages offered by this method, it is natural to explore how we can transform Hankel matrices, which do not inherently possess a circulant structure, into circulant matrices. In this subsection, we will delve into the specifics of this transformation process.

Transforming a Hankel matrix into a circulant matrix involves initially converting the Hankel matrix into a Toeplitz matrix. This conversion can be achieved by applying a permutation matrix, which reorders the columns of the Hankel matrix from left to right, thus forming a Toeplitz matrix. The permutation matrix P used in this process features a unique structure with all entries set to zero except for a single anti-diagonal filled with ones. While this permutation facilitates the transformation from Hankel to Toeplitz, it introduces unnecessary computational steps. However, for operations such as matrix-vector multiplication where only the outcome is needed, one can simply permute both the Hankel matrix and the vector concurrently, thereby preserving the result while effectively transforming the matrix's form to Toeplitz. Once in Toeplitz form, the matrix can be embedded into a larger circulant matrix to utilize efficient FFT computation methods. The embedding process involves combining the first column and the first row of the Toeplitz matrix to form the generating vector for the circulant matrix, which then becomes the first row of the new matrix C. Each subsequent row of C is a rightward cyclic shift of the row above. Figure 9 above serves as a simple example to illustrate the concept where a Toeplitz matrix is embedded into a circulant matrix.

Notably, for FFT-based computations, it is sufficient to focus solely on the generating vector of the circulant matrix; constructing the entire matrix explicitly is unnecessary. By focusing solely on the generating vector, we can bypass the redundant steps involved in creating and manipulating the full circulant matrix.

$$\begin{bmatrix} a & \mathrm{b} & \mathrm{c} \\ \mathrm{d} & \mathrm{a} & \mathrm{b} \\ \mathrm{e} & \mathrm{d} & \mathrm{a} \end{bmatrix} \xrightarrow{g=[a \quad b \quad c \quad d \quad e]} \begin{bmatrix} \mathrm{a} & \mathrm{b} & c & e & d \\ \mathrm{d} & a & b & c & e \\ \mathrm{e} & d & a & b & c \\ \mathrm{c} & e & d & a & b \\ \mathrm{b} & c & e & d & a \end{bmatrix}$$

Figure 9. On the left is a Toeplitz matrix, where the generating vector g is formed by combining the first row and first column of the Toeplitz matrix. The squared circulant matrix is created by right-shifting the generating row, while the Toeplitz matrix remains unchanged, marked by the red square on the right.

5.3. Implicit Element-Wise Blocked Matrix Multiplication

The input feature map is unrolled into a doubly block Hankel matrix, which naturally takes the form of a blocked matrix defined by its doubly Hankel structure. For multiplying this structure with the kernel matrix, a blocked matrix multiplication strategy is employed. Each Hankel block matrix multiplication is efficiently conducted using the Fast Fourier Transform (FFT). After transforming each Hankel block into the Fourier domain, the inherent linearity properties of the Discrete Fourier Transform (DFT) are utilized. This involves performing a direct summation of element-wise products within the Fourier domain for each block multiplication. Importantly, this method allows us to defer the Inverse Discrete Fourier Transform (IDFT) operations until after the summation for blocked matrix multiplication is complete. This strategy not only optimizes computational efficiency by reducing the number of IDFT operations required but also simplifies the entire processing workflow.

Traditional im2col-based convolution methods in convolutional neural networks typically involve unrolling the feature map either entirely or partially to perform matrix operations. This unrolling process leads to significant data redundancy, which in turn increases the memory footprint. However, since we have already established data correspondence in Section 4.6 and only require the generating vector for the circulant matrix, the need for a fully unrolled input matrix is eliminated, thus saving memory. Consequently, we can implement implicit blocked matrix multiplication without unrolling the matrix, where we utilize a unique indexing scheme that efficiently retrieves the necessary Fourier coefficients directly computed from the input feature maps. Figure 10 demonstrates the blocked matrix multiplication, where each block in the output is the summation of corresponding block matrix multiplications between the kernel matrix and the input matrix. Due to the Hankel matrix property, we do not need to compute the Discrete Fourier Transform (DFT) of identical blocks, as the blocks on the anti-diagonal are identical. Therefore, it suffices to compute the DFT for only the first column and last row of the doubly Hankel matrix. By eliminating redundant calculations for these identical Hankel blocks, this method significantly reduces the number of repetitive FFTs, thereby optimizing computational efficiency.

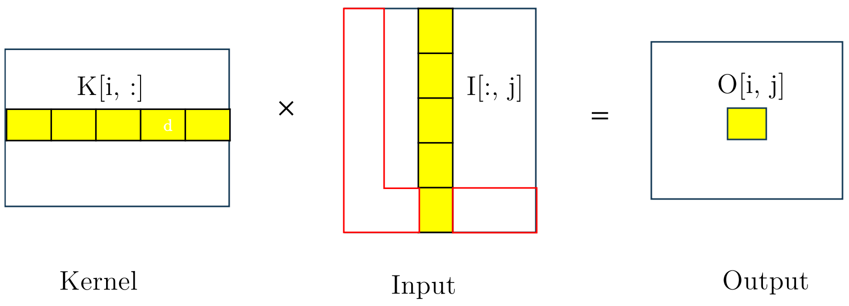


Figure 10. The process of blocked matrix multiplication involving the kernel matrix and the unrolled input matrix. The output O[i, j] is derived from the summation of products obtained by multiplying the corresponding blocks in K[i, :] and I[:, j]. Particularly, for the doubly blocked Hankel matrix in the unrolled input matrix, the first column and last row of Hankel blocks, marked by red rectangles are unique, while the other blocks are duplicates.

5.4. FFT Hermitian Symmetry and Gauss's Multiplication Formula

Hermitian symmetry, also known as conjugate symmetry, refers to a specific symmetric relationship in the Fourier transform of real-valued signals, where each coefficient is the complex conjugate of another coefficient at a symmetrically opposite position in the Fourier domain. In our implementation, the input data are real-valued, and the resulting frequency domain representation (Fourier coefficients) exhibits Hermitian symmetry. This means that for a discrete Fourier transform (DFT) of a real signal of length N, the Fourier coefficients will satisfy the following relationship: $X[k] = X[N-k]^*$ for all k, where $X[k]$ represents the Fourier coefficient at position k, and * denotes the complex conjugate. This implies that: (1) $X[0]$ and $X[N/2]$ (for even N) are both real values (no imaginary component). (2) The coefficients from $X[1]$ to $X[N/2-1]$ are the complex conjugates of the coefficients from $X[N-1]$ down to $X[N/2+1]$. In the implementation, we zero-pad both inputs and kernels to the next power of two to have the best performance of cuFFT. Because of Hermitian symmetry, we only need to store nearly half of the coefficients, effectively reducing the memory requirements by nearly half. Additionally, leveraging this symmetry, we can nearly halve the number of multiplications needed for element-wise multiplications. The second set of element-wise multiplications can be derived from the first by taking the complex conjugate.

Gauss's multiplication formula [53] offers a computationally efficient approach for multiplying complex numbers, which is particularly advantageous in the Fourier domain. Traditionally, the multiplication of two complex numbers, represented as $a+ib$ and $c+id$, requires four real multiplications. Gauss's method, however, reduces this requirement to three real multiplications. The algorithm computes three intermediate terms: $t_1 = c \times (a+b)$, $t_2 = a \times (d-c)$, and $t_3 = b \times (c+d)$. From these intermediates, the real and imaginary components of the product are subsequently derived from these intermediates as $t_1 - t_3$ for the real component and $t_1 + t_2$ for the imaginary component, respectively.

5.5. Entire Workflow Description

The proposed convolution method is implemented in four steps:

Step 1 Input transform. In this initial step, the convolution method leverages the structure of the input feature maps, where the generating vector of the circulant matrix is contained within each row. Accordingly, one-dimensional FFTs are applied to the generating vector.

Step 2 Kernel transform. The kernel matrix is segmented into tiles, with each tile corresponding to the width of the kernel. A one-dimensional FFT is then performed on each of these tiles. The cuFFT library, specifically its batch mode capability, is employed to execute these FFTs efficiently. This step is critical as it prepares the kernel by translating it into the Fourier domain.

Step 3 Element-wise multiplication. Following the kernel transformation, the next step involves performing element-wise multiplication within the predefined doubly block Hankel matrix. Element-wise multiplication is carried out within each Hankel block. This step leverages the structured nature of the data for efficient matrix multiplication.

Step 4 Inverse transform. The output from this operation typically yields a $1 \times 2Q$ matrix. However, for the purposes of this specific convolution method, only the $1 \times Q$ elements of this matrix are relevant. These elements represent the actual product of the convolution between the $1 \times V$vector and the $V \times Q$ Hankel matrix. The remainder of the output is discarded, as it does not contribute to the final desired outcome.

Our methodology relies heavily on FFT operations, for which we employ the cuFFT library, an NVIDIA proprietary tool designed specifically for this purpose. An important factor to consider for achieving optimal performance with the cuFFT library is its sensitivity to FFT size. Even slight variations in FFT dimensions can significantly impact performance due to the specific algorithms employed by cuFFT. To mitigate this sensitivity, we utilize input padding techniques. The cuFFT programming guide [54] suggests using power-of-two sizes for FFT dimensions to achieve optimal performance. Moreover, cuFFT operates as a "black box" due to its proprietary nature, meaning that its internal mechanisms are not accessible for modification or optimization by external users. This restriction can lead to less-than-optimal performance, especially when our implementation involves a limited range of power-of-two 1D FFTs, where the library's generalized approach may not be perfectly aligned with specific computational needs. As future work, we plan to develop our own custom FFT implementation to address these specific cases and potentially achieve further performance gains.

5.6. Arithmetic Complexity Comparison

The FFT-based approach from cuDNN and our fine-grained FFT approach share similar steps, including input transform, kernel transform, element-wise multiplication, and inverse transform. However, the primary difference lies in how the transformed data are processed: FFT-based convolution utilizes batched complex general matrix multiplication (Cgemm), whereas fine-grained FFT convolution involves matrix multiplication with an element-wise product. For analytical comparison, we can consider both methods under the umbrella of matrix multiplication. In this subsection, we will compare the arithmetic complexity of both convolution methods side by side.

Assuming we have inputs with dimensions (N, C, H, W) and kernels with dimensions (K, C, U, V), Table 2 outlines the arithmetic complexities associated with each step for both our fine-grained FFT and FFT-based convolution methods. The table indicates that the number of operations required for the input transform and the final inverse transform are comparable; however, our fine-grained FFT method requires fewer operations for the kernel transformation step. Conversely, our approach involves more operations during the matrix multiplication step. In FFT-based convolution, the kernel is zero-padded to match the dimensions of the input before performing the FFT. Consequently, the performance of the FFT-based method remains unaffected by the kernel's spatial dimensions, specifically parameters U and V. This approach may yield better performance with larger kernels. In contrast, the complexity of our method scales directly with the kernel size. This is because the size of the unrolled input matrix increases quadratically with the kernel size, making it more suited for convolutions with smaller kernels. A significant advantage of our fine-grained FFT method lies in its finer FFT granularity. Unlike conventional FFT approaches that perform a 2D FFT on each feature map with an FFT granularity of HW, our method leverages a finer granularity through the $2Q$ FFT. However, there is a notable trade-off in kernel size between the fine-grained FFT and FFT-based convolution methods; if the kernel size is large, the increased cost of matrix multiplication might negate the efficiency gains from the fine-grained FFT method.

Table 2. Arithmetic complexity comparison between FineGrainedFFT convolution and FFT-based convolution from cuDNN (refer to Table 1 for the notations used).

	FFT-Based	**FineGrainedFFT**
Input transform	$2W^2 \cdot \log W \cdot N \cdot C$	$2W \cdot \log 2W \cdot Q \cdot N \cdot C$
Kernel transform	$2W^2 \cdot \log W \cdot K \cdot C$	$2W \cdot \log 2W \cdot K \cdot C \cdot V$
Matrix multiplication	$W^2 \cdot N \cdot K \cdot C$	$2W \cdot Q \cdot N \cdot K \cdot C \cdot V$
Inverse transform	$2W^2 \cdot \log W \cdot N \cdot K$	$2W \cdot \log 2W \cdot Q \cdot N \cdot K$

5.7. Memory Consumption Comparison

Our fine-grained FFT technique avoids the full unrolling of input feature maps, which is a common practice in the im2col+GEMM method, leading to more efficient memory usage. In this analysis, we compare the memory footprints of our method with that of the im2col+GEMM approach to highlight the improvements in memory efficiency.

The im2col+GEMM method involves a complete unrolling of the input feature maps, where, as the kernel traverses the feature maps, it transforms the local patches into columns, which are then aligned side by side to form an unrolled matrix. This process leads to a significant increase in memory requirements, as the space needed for these unrolled feature maps grows quadratically with the size of the kernel. In contrast, our fine-grained FFT method does not require the full unrolling of the input feature maps. This is because the original feature map already contains the necessary elements for generating vectors for matrix-vector multiplication using FFTs. To efficiently compute FFTs in our approach, it is only necessary to pad each row in the feature maps to $2Q$. More precisely, the padded size is calculated using the NextPowerTwo(2Q), a function that determines the nearest power of two. For an input with the format NCHW, the im2col+GEMM method requires a memory consumption of $NCUVRQ$, whereas our method necessitates only $2NCHQ$ for its memory footprint. Furthermore, due to the Hermitian symmetry of the Fourier transform of a real-valued input, we only need to store half of the complex entries, each of which is twice the size of a float type. In summary, our method saves nearly $(UVR)/(2H)$ times more memory than the im2col+GEMM method. To make the comparison presented in this subsection more clear, let us consider an illustrative example where the kernel size $(U \times V)$ is set to 3×3 and the input height (H) is 254. According to Equation (1), which determines output dimensions based on input and kernel sizes, the output height (R) is 254. In the expression $UVR/2H$, both H and R are eliminated during the calculation, resulting in $UV/2$, which gives a value of 4.5.

5.8. Auto-Tuning

Auto-tuning, also known as automatic performance tuning, optimizes software for efficient operation on specific hardware platforms. Particularly useful in GPU programming, our implementation employs auto-tuning to dynamically select optimal CUDA thread and block parameters. This selection process considers the specific constraints imposed by the input settings and available hardware resources. The identified optimal configuration can be stored locally, enabling reuse for subsequent executions with similar input configurations. Our auto-tuning strategy is crucial, especially as the fine-grained FFT approach it supports comprises four distinct steps, each benefiting from tailored optimization of CUDA parameters. For example, during the third step of element-wise multiplication, auto-tuning determines the best thread block size and number of blocks by exploring various combinations and executing only the most promising ones. It avoids an exhaustive search, reducing the time it takes to find the optimal setup. We implement the autotuner through a code generator that produces parameterized kernel variants, which are then compiled, executed, and benchmarked to determine the best option. Overall, auto-tuning provides a speedup of approximately 5% compared to versions without auto-tuning.

6. Evaluation and Performance Analysis

6.1. Experimental Methodology

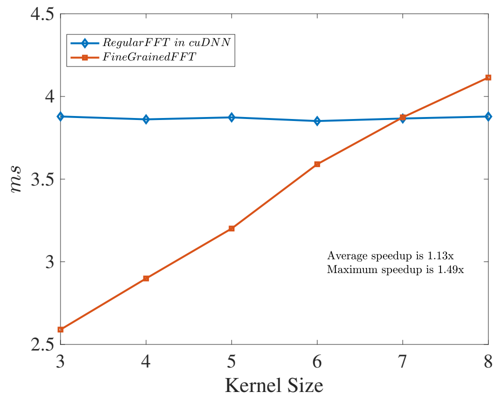
To establish clarity in our discussion, let us define the naming convention used here: our method is termed "FineGrainedFFT", distinguishing it from the FFT method provided by cuDNN, which we refer to as "RegularFFT". The evaluation comprises two levels: At the low level, we compare it with other current convolution methods head-to-head. At the high level, we replace the convolution methods used in a leading deep learning framework with our FineGrainedFFT method and compare the performance on real-world benchmarks before and after the replacement. Specifically, we evaluate the proposed method across four dimensions: (1) accuracy of result. (2) Kernel-level performance comparison. We compare our performance with NVIDIA's cuDNN library [20], which features a variety of state-of-the-art convolution implementations. We utilize the cuFFT [54] library to compute FFTs and employ both a synthetic benchmark and the 2017 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) for an object localization benchmark, the latter being widely used as the input for performance evaluation in a prior study on CNNs. Parameters are organized into a 2-tuple (U, N), with commonly used benchmarking values assigned to other parameters (K, C, H). For kernel timing comparisons, the initial cuFFT library call, which incurs significant setup costs, is excluded from total execution time through a preliminary warmup call that isolates these costs. Each performance measurement is averaged over five runs. (3) Application-scenario performance comparison. We replace Caffe's [21] convolution method and measure inference times for several synthetic CNNs, comparing these against cuDNN in Caffe. In addition, we measure the layer-wise execution times for each convolution layer. (4) Performance profiling is conducted to analyze the sources of performance improvements. Both RegularFFT and FineGrainedFFT approaches invoke FFTs and element-wise multiplication, and the theoretical arithmetic complexity is analyzed in Section 5.6. The empirical timings provide the details of the performance breakdown of the algorithms, thus explaining the source of the performance gain of our optimization. To demonstrate that our algorithm's performance gains are consistent across different hardware configurations, we ran experiments on two Nvidia GPUs: a Titan Xp (Pascal architecture) and a Tesla T4 (Turing architecture). The Titan Xp utilizes cuDNN v7.1, while the Tesla T4 employs cuDNN v8.9. The version of Caffe used is 1.0. For our experiments, the CPU primarily functions as a command processor, having a negligible effect on overall performance.

6.2. Accuracy

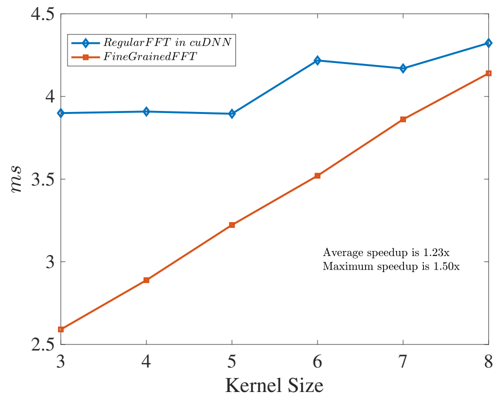
Initially, it is crucial to assess the accuracy of our FineGrained FFT method, as it provides a quantitative metric to measure its effectiveness and reliability. We conduct a comparative analysis with the im2col+MM approach from cuDNN, which serves as the ground truth. For parameter configurations (U, K) set to (3, 10), (5, 32), and (4, 10), the absolute element errors are found to be 4.73×10^{-11}, 3.00×10^{-11}, and 2.38×10^{-11}, respectively. These results indicate that the error magnitude is approximately 10^{-11}, demonstrating that our optimization technique effectively preserves the numerical integrity of the convolution process. This preservation of accuracy is critical, especially in applications where even minor deviations can significantly impact the overall system performance. Thus, our method ensures that the enhancement in efficiency does not compromise accuracy, underscoring its robustness and suitability for practical applications.

6.3. Kernel Performance Comparison

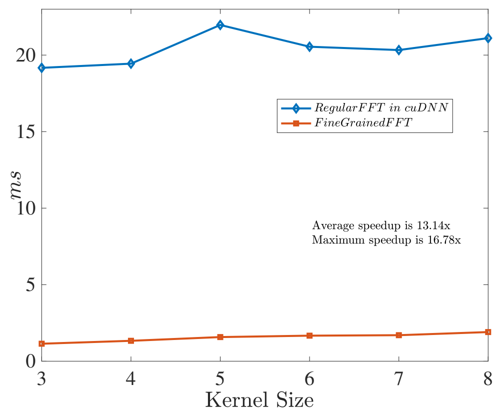
To rigorously assess the kernel-level performance of our proposed method, we directly measure pure GPU kernel execution times and compare them against the FFT-based and im2col+GEMM convolution implementations within cuDNN. This comparative analysis is conducted across varying kernel and batch sizes to comprehensively evaluate the strengths and weaknesses of each approach under different parameter settings. In our analysis, illustrated in Figure 11, we compare the FineGrained FFT algorithm with the Regular FFT algorithm using both a synthetic benchmark and the 2017 ILSVRC object localization benchmark on Nvidia Titan Xp and Tesla T4 GPUs, respectively. For the synthetic benchmark, we generate random input data and kernels sampled uniformly from the range $[-1, 1]$. The results indicate that the execution time for the RegularFFT convolution remains relatively constant and unaffected by changes in kernel size. This stability is due to the method's use of zero padding to match the kernel size to the feature map size, making the kernel size nearly irrelevant to its performance. Such consistency is maintained across different GPU architectures. Conversely, the performance of our proposed method declines as the kernel size increases, primarily due to its direct dependency on the kernel size, despite our approach not requiring full matrix unrolling. This trend is consistent in both Figure 11a,b. Given that our experimental measurements are obtained at approximately 4 milliseconds, we notice variations in the outcomes across various datasets. These variations fell within the reasonable fluctuation range, leading to an intersection in Figure 11a but not in Figure 11b. While the curves representing FineGrained FFT convolution in both Figure 11c,d appear visually straight, a closer look reveals that they have a small but non-zero slope. This indicates that the execution time actually grows with increasing kernel size, despite being seemingly constant. This behavior is due to the larger values of regular FFT, and FineGrained FFT seems small by comparison. It is worth noting in the cuDNN release notes that convolutions in cuDNN version 8.9.6, which is employed in Tesla T4, may experience performance regressions [55], resulting in larger gaps between curves in Figure 11c,d. Overall, our FineGrained FFT convolution consistently delivers high performance. In Figure 12, we compare the FineGrained FFT algorithm with the im2col+GEMM algorithm from cuDNN. As the kernel size increases from 5 to 30, the execution time for both algorithms rises; however, the im2col+GEMM algorithm exhibits a more pronounced increase. Our approach achieves higher performance primarily due to its strategy of not fully unrolling the feature maps. In contrast, the im2col+GEMM method requires a quadratic growth in unrolled matrix size with increasing kernel size. By avoiding a full unrolling of feature maps, our approach minimizes memory footprint, allowing it to achieve better performance.



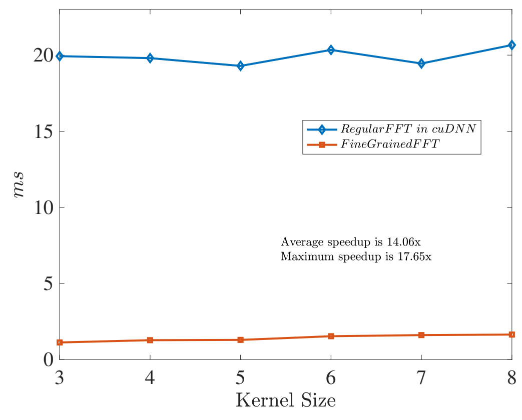
(**a**) Synthetic benchmark on Titan Xp



(**b**) ILSVRC2017 benchmark on Titan Xp



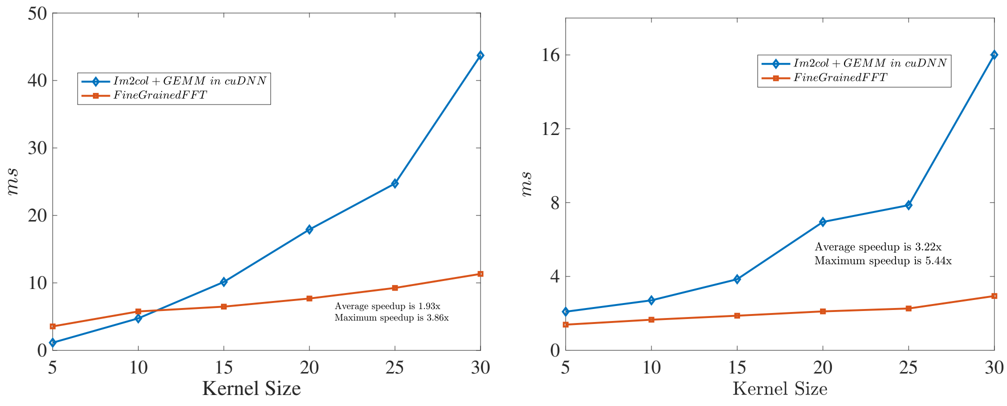
(**c**) Synthetic benchmark on Tesla T4



(**d**) ILSVRC2017 benchmark on Tesla T4

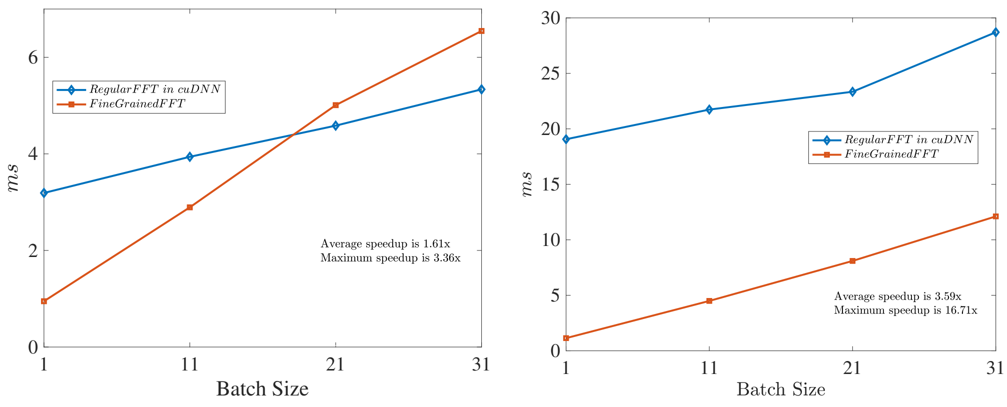
Figure 11. RegularFFT and FineGrainedFFT performance comparison as the kernel size varies from 3 to 8.

We proceed to evaluate performance across different batch sizes, as depicted in Figure 13. At a batch size of 20, as shown in Figure 13a, the two curves intersect, indicating that the FineGrained FFT surpasses the regular FFT for smaller batch sizes. In the case of regular FFT, the convolution operation is transformed into point-wise multiplication in the spatial domain. This process is efficiently executed as batched matrix multiplication, which is particularly well-suited for the highly optimized cuBLAS library. As the batch size increases, regular FFT matrix multiplication begins to outperform FineGrained FFT. Consequently, for FineGrained FFT, an increase in batch size impacts element-wise multiplication negatively, potentially negating the benefits provided by FFT. Across Figures 11 and 13a, the FineGrained FFT exhibits consistently better performance than the regular FFT on Tesla T4. Meanwhile, the FineGrained FFT convolution performs particularly well for small kernels and batches, demonstrating an advantage over the RegularFFT approach in Figures 11c,d and 13b. Overall, when considering the parameter space of convolutions, FineGrained FFT convolution demonstrates particular advantages for scenarios involving small kernel sizes and batch sizes.



(**a**) Titan Xp (**b**) Tesla T4

Figure 12. Im2col+GEMM and FineGrainedFFT performance comparison on ILSVRC2017 benchmark.



(**a**) Titan Xp (**b**) Tesla T4

Figure 13. Performance comparison of RegularFFT and FineGrainedFFT with varying batch sizes (1 to 31) on the ILSVRC2017 benchmark.

6.4. Performance in Applications

Caffe is widely recognized as a popular platform for creating deep learning applications. In our experiment, we replace the convolutional implementation in Caffe with our FineGrained approach.

Synthetic Benchmark: We construct four convolutional neural networks (CNNs), each consisting of five convolutional layers, with configurations detailed in Table 3. These networks incorporate pooling and rectified linear unit (ReLU) layers, concluding with a fully connected layer with 10 outputs for prediction. The batch sizes used are 5, 10, 1, and 3, respectively. As illustrated in Figure 3, our FineGrained FFT convolution outperforms the Regular FFT convolution across all tested configurations during a single CNN inference iteration. It achieves speed improvements of 2.12×, 1.19×, 1.92×, and 1.46× over the Regular FFT convolution.

SqueezeNet Benchmark: SqueezeNet [56] is a deep convolutional neural network (CNN) architecture known for its efficiency and compact size. The core building block of SqueezeNet is the "Fire" module, which consists of two types of layers: squeeze layers and expand layers. Squeeze layers are compact 1 × 1 convolutional layers designed to reduce the number of feature maps, while expand layers utilize larger 3 × 3 and 1× 1 convolutions to expand the feature maps. Due to its architectural efficiency, SqueezeNet significantly reduces the number of parameters compared to larger networks, making it highly suitable for resource-constrained environments such as embedded systems and mobile applications. Given the exceptional performance of our FineGrained FFT algorithm for small kernel sizes, SqueezeNet emerges as a highly suitable network architecture. When the input size of (N, C, H, W) is (1, 3, 254, 254), our FineGrained FFT algorithm outperforms the Regular FFT algorithm in terms of inference speed, demonstrating a running time of 40.54 ms compared to the Regular FFT's 44.31 ms. This results in a performance improvement of approximately 9 percent.

VGG-16 Benchmark: We also assess the effectiveness of our method using a variant of the VGG-16 model [34]. VGG-16 is a popular benchmark for CNNs due to its improvements over previous CNN architectures. It achieves this by incorporating thirteen convolutional layers, each using 3 × 3 filters. In our evaluation, we compare the performance of this VGG-16 variant using two convolution implementations: our FineGrained FFT convolution and the Regular FFT convolution. We configure both approaches to utilize 38 kernels and a batch size of 3. The FineGrained FFT version achieves a faster forward-only inference time of 65.96 ms, demonstrating a 1.25× speedup compared to the Regular FFT-based convolution, which takes 82.76 milliseconds.

Table 3. Configuration of four synthetic networks with execution time for each convolution layer. Each network is composed of convolutional layers labeled ConvX-Y, where X denotes the kernel size and Y denotes the number of kernels. The execution times for each convolutional layer are measured in milliseconds (ms).

	Conv. Layer	**RegularFFT**	**FineGrainedFFT**	**Speedup**
Network1	Conv3-10	3.86	1.83	2.11×
	Conv3-5	4.43	1.29	3.43×
	Conv3-8	4.81	1.44	3.34×
	Conv3-7	4.09	1.45	2.82×
	Conv3-10	4.05	1.74	2.33×
Network2	Conv3-5	3.93	2.81	1.40×
	Conv4-10	4.55	4.97	0.92×
	Conv3-5	6.71	4.13	1.62×
	Conv5-5	4.18	3.47	1.20×
	Conv3-5	3.93	2.98	1.32×
Network3	Conv3-10	3.67	2.01	1.84×
	Conv3-8	4.50	1.52	2.96×
	Conv5-5	4.49	1.35	3.33×
	Conv3-10	4.05	1.49	2.72×
	Conv3-5	4.28	1.37	3.12×
Network4	Conv3-5	3.56	2.64	1.34×
	Conv3-7	3.34	2.54	1.31×
	Conv3-10	4.36	2.43	1.79×
	Conv3-8	3.63	3.21	1.13×
	Conv3-7	3.73	2.13	1.75×

To understand the source of the performance improvement observed in VGG-16, we perform a detailed layer-wise comparison between our FineGrained FFT and Regular FFT implementation. This analysis focuses on the seven layers within VGG-16 that account for roughly 60% of the overall execution time. The results, illustrated in Figure 14, demonstrate a layer-by-layer comparison. On average, FineGrained FFT outperforms Regular FFT by a factor of 1.7, with a maximum speedup of 2.86.

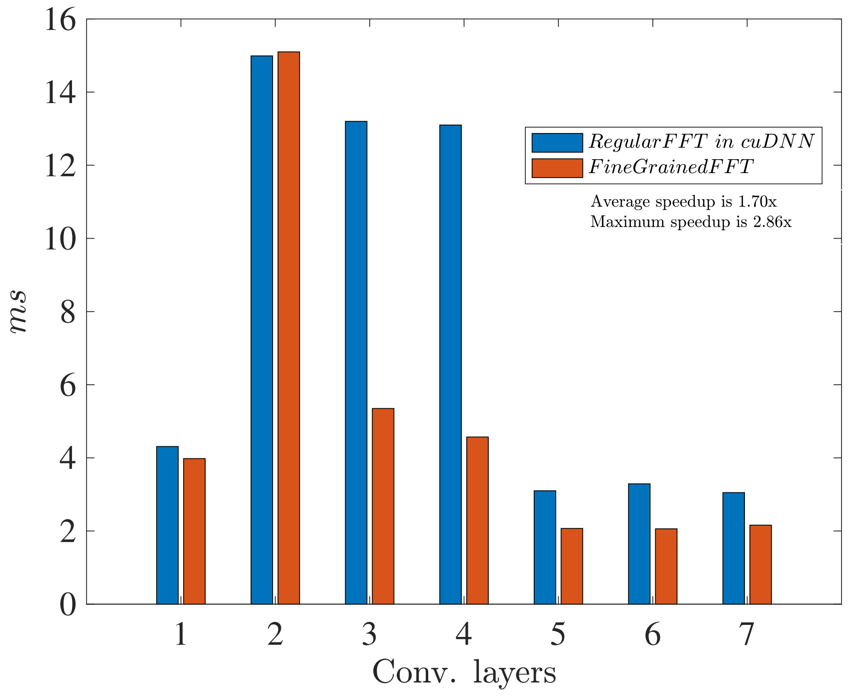


Figure 14. Layer-wise performance comparison between FineGrainedFFT and RegularFFT convolutions across the seven most computationally intensive layers of the VGG-16 network.

6.5. Performance Profiling Analysis

To gain deeper insights into the performance improvements achieved by our algorithm, we employ the nvprof profiling tool to analyze GPU kernels. Specifically, we compare the execution times of the FFTs and matrix multiplications for both algorithms, presenting the results in Figure 15. This comprehensive performance profiling offers a detailed breakdown of execution times, enabling us to identify the steps that contribute most significantly to the overall performance gain. In RegularFFT convolution, both the FFT and matrix multiplication steps exhibit relatively constant execution times as the kernel size increases. This is because the kernel is zero-padded to match the size of the input feature map. This padding strategy leads to a constant amount of computations, independent of the kernel size. Conversely, the time it takes for the matrix multiplication in FineGrained FFT grows as the size of the kernel size increases. This is because our method is dependent on an unrolled matrix similar to im2col. As the kernel size grows, this matrix becomes larger, leading to more computations. Additionally, the matrix multiplication for RegularFFT is essentially a series of batched matrix multiplications that involve transposition operations to align the tensors for the multiplication process, adding to the computational steps. Consequently, matrix multiplication in RegularFFT has higher execution times than FineGrained convolution. The FFT step in the FineGrained FFT generally maintains constant execution time, as the FFT size is tailored to the nearest power of two that corresponds to the input size.

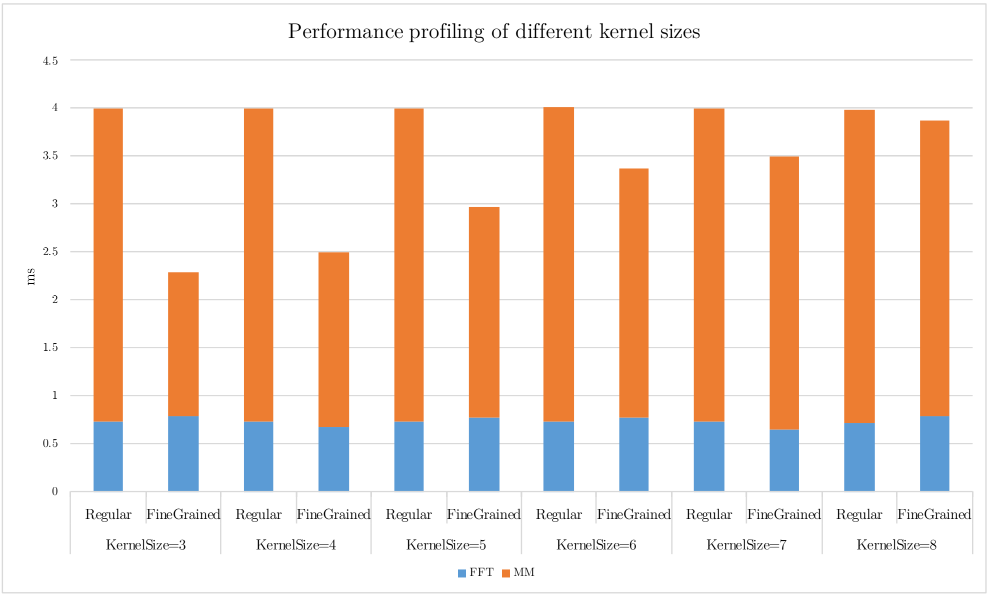


Figure 15. Profiling results for varying kernel sizes, illustrating the breakdown of execution time into FFT (Fast Fourier Transform) and MM (Matrix Multiplication).

7. Conclusions

In this study, we conducted a comprehensive analysis of data redundancy that arises from the im2col process, which is important in high-performance implementations of convolution. Our investigation identified two novel forms of redundancy: intra-row and inter-row, observed within im2col+GEMM convolution operations. These insights have driven us to formulate a more mathematically efficient representation of the data structure within convolutions, leading to the development of a doubly block Hankel matrix data pattern description. This paper presents a theoretical complexity analysis that compares our method with the FFT convolution used in NVIDIA's cuDNN library. Empirical results validate the theoretical analysis, demonstrating that our FineGrained FFT convolution consistently surpasses the FFT-based method across most convolution parameter settings. Specifically, our approach achieves an average speedup of 14 times and a maximum of 17 times compared to the regular FFT convolution, and an average speedup of 3 times and a maximum of 5 times over the im2col+GEMM method used in cuDNN.

Our efforts contribute to the diverse array of convolution techniques in CNNs. Furthermore, given the absence of a "one-size-fits-all" convolution implementation across the entire parameter space, our work enhances the overall performance of convolutions. By contributing FineGrained FFT convolutions with performance gain, we increase the envelope of optimal options available across the parameter space. In addition to enhancing convolution performance, one avenue for future work is the adaptive selection of optimal convolution algorithms based on specific configuration parameters. Alternatively, convolution in CNNs might be further optimized by leveraging the double Hankel matrix representation of the unrolled im2col matrix from a different perspective. Through this approach, the convolution operation is reinterpreted as a polynomial multiplication problem.

Author Contributions: Conceptualization, Y.Z. and X.L.; methodology, H.X. and S.J.; software, Y.Z. and F.L.; validation, F.L., H.X. and S.J.; investigation, H.X.; writing—original draft preparation, Y.Z. and S.J.; writing—review and editing, Y.Z., X.L. and S.J. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The original contributions presented in the study are included in the article, further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Tan, M.; Le, Q. Efficientnet: Rethinking model scaling for convolutional neural networks. In Proceedings of the International Conference on Machine Learning, Long Beach, CA, USA, 9–15 June 2019; pp. 6105–6114.
2. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep Residual Learning for Image Recognition. In Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, 27–30 June 2016; pp. 770–778. [CrossRef]
3. Deng, J.; Dong, W.; Socher, R.; Li, L.; Li, K.; Li, F. ImageNet: A large-scale hierarchical image database. In Proceedings of the 2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), Miami, FL, USA, 20–25 June 2009; pp. 248–255. [CrossRef]
4. Ren, S.; He, K.; Girshick, R.; Sun, J. Faster r-cnn: Towards real-time object detection with region proposal networks. *Adv. Neural Inf. Process. Syst.* **2015**, *28*, 1137–1149. [CrossRef] [PubMed]
5. Redmon, J.; Divvala, S.; Girshick, R.; Farhadi, A. You only look once: Unified, real-time object detection. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 779–788.
6. Tan, M.; Pang, R.; Le, Q.V. Efficientdet: Scalable and efficient object detection. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, Seattle, WA, USA, 13–19 June 2020; pp. 10781–10790.
7. Ronneberger, O.; Fischer, P.; Brox, T. U-net: Convolutional networks for biomedical image segmentation. In Proceedings of the Medical Image Computing and Computer-Assisted Intervention—MICCAI 2015: 18th International Conference, Munich, Germany, 5–9 October 2015; Proceedings, Part III 18; Springer: Berlin/Heidelberg, Germany, 2015; pp. 234–241.
8. Chen, L.C.; Papandreou, G.; Kokkinos, I.; Murphy, K.; Yuille, A.L. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE Trans. Pattern Anal. Mach. Intell.* **2017**, *40*, 834–848. [CrossRef] [PubMed]
9. Karras, T.; Laine, S.; Aila, T. A style-based generator architecture for generative adversarial networks. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, Long Beach, CA, USA, 15–20 June 2019; pp. 4401–4410.
10. Brock, A.; Donahue, J.; Simonyan, K. Large scale GAN training for high fidelity natural image synthesis. *arXiv* **2018**, arXiv:1809.11096.
11. Bojarski, M.; Del Testa, D.; Dworakowski, D.; Firner, B.; Flepp, B.; Goyal, P.; Jackel, L.D.; Monfort, M.; Muller, U.; Zhang, J.; et al. End to end learning for self-driving cars. *arXiv* **2016**, arXiv:1604.07316.
12. Chen, C.; Seff, A.; Kornhauser, A.L.; Xiao, J. DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving. In Proceedings of the 2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, 7–13 December 2015; pp. 2722–2730. [CrossRef]
13. Hadjis, S.; Abuzaid, F.; Zhang, C.; Ré, C. 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, 31 May–4 June 2015; Katsifodimos, A., Ed.; ACM: New York, NY, USA, 2015; pp. 2:1–2:4. [CrossRef]
14. Cong, J.; Xiao, B. Minimizing Computation in Convolutional Neural Networks. In Proceedings of the Artificial Neural Networks and Machine Learning—ICANN 2014, Hamburg, Germany, 15–19 September 2014; Springer International Publishing: Cham, Switzerland, 2014; pp. 281–290.
15. Park, H.; Kim, D.; Ahn, J.; Yoo, S. 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, PA, USA, 1–7 October 2016; ACM: New York, NY, USA, 2016; pp. 33:1–33:10. [CrossRef]
16. Vasilache, N.; Johnson, J.; Mathieu, M.; Chintala, S.; Piantino, S.; LeCun, Y. Fast Convolutional Nets with fbfft: A GPU Performance Evaluation. In Proceedings of the 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, 7–9 May 2015.
17. Jia, Y. Learning Semantic Image Representations at a Large Scale. Ph.D. Thesis, University of California, Berkeley, CA, USA, 2014.
18. Li, X.; Zhang, G.; Huang, H.H.; Wang, Z.; Zheng, W. Performance Analysis of GPU-Based Convolutional Neural Networks. In Proceedings of the 45th International Conference on Parallel Processing, ICPP 2016, Philadelphia, PA, USA, 16–19 August 2016; pp. 67–76. [CrossRef]
19. Perkins, H. cltorch: A Hardware-Agnostic Backend for the Torch Deep Neural Network Library, Based on OpenCL. *arXiv* **2016**, arXiv:1606.04884.
20. Chetlur, S.; Woolley, C.; Vandermersch, P.; Cohen, J.; Tran, J.; Catanzaro, B.; Shelhamer, E. cuDNN: Efficient Primitives for Deep Learning. *arXiv* **2014**, arXiv:1410.0759.
21. Jia, Y.; Shelhamer, E.; Donahue, J.; Karayev, S.; Long, J.; Girshick, R.B.; Guadarrama, S.; Darrell, T. Caffe: Convolutional Architecture for Fast Feature Embedding. In Proceedings of the ACM International Conference on Multimedia, MM '14, Orlando, FL, USA, 3–7 November 2014; Hua, K.A., Rui, Y., Steinmetz, R., Hanjalic, A., Natsev, A., Zhu, W., Eds.; ACM: New York, NY, USA, 2014; pp. 675–678. [CrossRef]
22. Yu, D.; Eversole, A.; Seltzer, M.L.; Yao, K.; Guenter, B.; Kuchaiev, O.; Seide, F.; Wang, H.; Droppo, J.; Huang, Z.; et al. An introduction to computational networks and the computational network toolkit (invited talk). In Proceedings of the INTERSPEECH 2014, 15th Annual Conference of the International Speech Communication Association, Singapore, 14–18 September 2014.
23. Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. TensorFlow: A System for Large-Scale Machine Learning. In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, Savannah, GA, USA, 2–4 November 2016; OSDI '16, pp. 265–283.

24. Al-Rfou, R.; Alain, G.; Almahairi, A.; Angermueller, C.; Bahdanau, D.; Ballas, N.; Bastien, F.; Bayer, J.; Belikov, A.; Belopolsky, A.; et al. Theano: A Python framework for fast computation of mathematical expressions. *arXiv* **2016**, arXiv:1605.02688.
25. Collobert, R.; Kavukcuoglu, K.; Farabet, C. Torch7: A Matlab-like Environment for Machine Learning. In Proceedings of the BigLearn, NIPS Workshop, Granada, Spain, 12–14 December 2011.
26. Shen, M.; Wang, J.; Du, H.; Niyato, D.; Tang, X.; Kang, J.; Ding, Y.; Zhu, L. Secure Semantic Communications: Challenges, Approaches, and Opportunities. *IEEE Netw.* **2024**, *38*, 197–206. [CrossRef]
27. Sabir, D.; Hanif, M.A.; Hassan, A.; Rehman, S.; Shafique, M. TiQSA: Workload Minimization in Convolutional Neural Networks Using Tile Quantization and Symmetry Approximation. *IEEE Access* **2021**, *9*, 53647–53668. [CrossRef]
28. Gysel, P.; Motamedi, M.; Ghiasi, S. Hardware-oriented Approximation of Convolutional Neural Networks. *arXiv* **2016**, arXiv:1604.03168.
29. Limonova, E.; Sheshkus, A.; Ivanova, A.A.; Nikolaev, D. Convolutional Neural Network Structure Transformations for Complexity Reduction and Speed Improvement. *Pattern Recognit. Image Anal.* **2018**, *28*, 24–33. [CrossRef]
30. Cintra, R.; Duffner, S.; Garcia, C.; Leite, A. Low-Complexity Approximate Convolutional Neural Networks. *IEEE Trans. Neural Netw. Learn. Syst.* **2018**, *29*, 5981–5992. [CrossRef] [PubMed]
31. Mathieu, M.; Henaff, M.; LeCun, Y. Fast Training of Convolutional Networks through FFTs. In Proceedings of the 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, 14–16 April 2014.
32. Chellapilla, K.; Puri, S.; Simard, P. High Performance Convolutional Neural Networks for Document Processing. In Proceedings of the Tenth International Workshop on Frontiers in Handwriting Recognition, La Baule, France, 23–26 October 2006; Lorette, G., Ed.; Université de Rennes 1: La Baule, France, 2006.
33. Lavin, A.; Gray, S. Fast Algorithms for Convolutional Neural Networks. In Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, 27–30 June 2016; pp. 4013–4021. [CrossRef]
34. Simonyan, K.; Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. In Proceedings of the 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, 7–9 May 2015.
35. Ioffe, S.; Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6–11 July 2015; JMLR Workshop and Conference Proceedings; Bach, F.R., Blei, D.M., Eds.; Volume 37, pp. 448–456.
36. Tollenaere, N.; Iooss, G.; Pouget, S.; Brunie, H.; Guillon, C.; Cohen, A.; Sadayappan, P.; Rastello, F. Autotuning convolutions is easier than you think. *ACM Trans. Archit. Code Optim.* **2023**, *20*, 1–24. [CrossRef]
37. Zhang, Y.; Li, X. Fast Convolutional Neural Networks with Fine-Grained FFTs. In Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques, New York, NY, USA, 3–7 October 2020; PACT '20, pp. 255–265. [CrossRef]
38. Goodfellow, I.; Bengio, Y.; Courville, A. *Deep Learning*; The MIT Press: Cambridge, MA, USA, 2016.
39. Krizhevsky, A. cuda-Convnet: High-Performance c++/Cuda Implementation of Convolutional Neural Networks. 2012. Available online: https://github.com/akrizhevsky/cuda-convnet2 (accessed on 23 January 2019).
40. Georganas, E.; Avancha, S.; Banerjee, K.; Kalamkar, D.; Henry, G.; Pabst, H.; Heinecke, A. Anatomy of high-performance deep learning convolutions on simd architectures. In Proceedings of the SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, 11–16 November 2018; pp. 830–841.
41. Lavin, A. maxDNN: An Efficient Convolution Kernel for Deep Learning with Maxwell GPUs. *arXiv* **2015**, arXiv:1501.06633.
42. Gray, S. Maxas: Assembler for Nvidia Maxwell Architecture. 2014. Available online: https://github.com/NervanaSystems/maxas (accessed on 8 February 2019).
43. Oyama, Y.; Nomura, A.; Sato, I.; Nishimura, H.; Tamatsu, Y.; Matsuoka, S. Predicting statistics of asynchronous SGD parameters for a large-scale distributed deep learning system on GPU supercomputers. In Proceedings of the 2016 IEEE International Conference on Big Data, BigData 2016, Washington, DC, USA, 5–8 December 2016; pp. 66–75. [CrossRef]
44. Vasudevan, A.; Anderson, A.; Gregg, D. Parallel Multi Channel convolution using General Matrix Multiplication. In Proceedings of the 2017 IEEE 28th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), Seattle, WA, USA, 10–12 July 2017; pp. 19–24. [CrossRef]
45. Wang, Q.; Mei, S.; Liu, J.; Gong, C. Parallel convolution algorithm using implicit matrix multiplication on multi-core CPUs. In Proceedings of the 2019 IEEE International Joint Conference on Neural Networks (IJCNN), Budapest, Hungary, 14–19 July 2019; pp. 1–7.
46. Zhao, Y.; Lu, J.; Chen, X. An Accelerator Design Using a MTCA Decomposition Algorithm for CNNs. *Sensors* **2020**, *20*, 5558. [CrossRef] [PubMed]
47. Highlander, T.; Rodriguez, A. Very Efficient Training of Convolutional Neural Networks using Fast Fourier Transform and Overlap-and-Add. *arXiv* **2016**, arXiv:1601.06815.
48. Abtahi, T.; Shea, C.; Kulkarni, A.; Mohsenin, T. Accelerating Convolutional Neural Network With FFT on Embedded Hardware. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2018**, *26*, 1737–1749. [CrossRef]
49. Pratt, H.; Williams, B.M.; Coenen, F.; Zheng, Y. FCNN: Fourier Convolutional Neural Networks. In Proceedings of the ECML PKDD 2017, Skopje, Macedonia, 18–22 September 2017; pp. 786–798. [CrossRef]
50. Winograd, S. *Arithmetic Complexity of Computations*; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 1980.
51. Partington, J. *An Introduction to Hankel Operators*; London Mathematical Society Student Texts; Cambridge University Press: Cambridge, UK, 1988.

52. Gray, R.M. Toeplitz and circulant matrices: A review. *Found. Trends® Commun. Inf. Theory* **2006**, *2*, 155–239. [CrossRef]
53. MacLaren, M.D. The art of computer programming. Volume 2: Seminumerical algorithms (Donald E. Knuth). *SIAM Rev.* **1970**, *12*, 306–308. [CrossRef]
54. NVIDIA. Programming Guide, CUSPARSE, CUBLAS, and CUFFT Library User Guides. Available online: https://docs.nvidia.com/cuda/ (accessed on 15 February 2019).
55. NVIDIA. cuDNN Release Notes. 2023. Available online: https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-897/release-notes/ (accessed on 9 April 2024).
56. Iandola, F.N.; Moskewicz, M.W.; Ashraf, K.; Han, S.; Dally, W.J.; Keutzer, K. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *arXiv* **2016**, arXiv:1602.07360.