

**FAST CONVOLUTIONAL NEURAL NETWORKS
ON GRAPHICS PROCESSING UNITS**

by

Yulin Zhang

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering

Fall 2019

© 2019 Yulin Zhang
All Rights Reserved

**FAST CONVOLUTIONAL NEURAL NETWORKS
ON GRAPHICS PROCESSING UNITS**

by

Yulin Zhang

Approved: _____

Kenneth E. Barner, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: _____

Levi T. Thompson, Ph.D.
Dean of the College of Engineering

Approved: _____

Douglas J. Doren, Ph.D.
Interim Vice Provost for Graduate and Professional Education and
Dean of the Graduate College

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____
Xiaoming Li, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____
Hui Fang, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____
Xiugang Wu, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____
Yuanfang Chen, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

First and foremost, I want to gratefully thank my advisor Professor Xiaoming Li. Without his guidance, support, and help, I would not be able to reach here. Over the years throughout my Ph.D. studies, Professor Li has taught me not only on how to conduct primary and rigorous research, more importantly, his knowledgeable insight into research problems, immense dedication towards work, and meticulous understanding for every aspect of work have influenced me and I will take them with me for my future career.

I am fortunate to work with Professor John Cavazos and Professor Hui Fang. The knowledge and skills I have learnt from them have improved my professional skills. It has been a great experience working with them. I would like to thank other dissertation committee members, Professor Xiugang Wu and Dr. Yuanfang Chen, for their time and helpful suggestions. I also would like to thank Dr. Hao Wu for help on the information retrieval project.

Finally, I want to thank my family for their support and always believing in me in good times or hard times. I am deeply indebted to them for their unconditional love. This dissertation is dedicated to them.

TABLE OF CONTENTS

LIST OF TABLES	viii
LIST OF FIGURES	ix
ABSTRACT	xii
Chapter	
1 INTRODUCTION	1
2 BACKGROUND	5
2.1 Convolutional Neural Network	5
2.1.1 Convolutional Neural Network Architecture	5
2.1.2 Convolutional Layers	6
2.1.3 Sub-sampling and Fully-connected Layers	8
2.1.4 Representative CNN Models	9
2.1.5 Training and Inference	11
2.2 Convolution in CNNs	12
2.2.1 2D Convolution	12
2.2.2 CNN Convolution	13
2.2.3 Convolution in Backward Pass	15
2.3 Deep Learning Frameworks	17
2.3.1 Caffe	18
2.3.2 Data Layout	18
2.3.3 ImageNet Benchmark	19
2.4 Graphics Processing Units (GPUs)	20
2.5 Circulant Matrices	21

3	EXISTING CONVOLUTION ALGORITHMS	26
3.1	Direct Convolution	26
3.2	Im2col+GEMM Convolution	27
3.3	FFT-based Convolution	30
3.4	Winograd Convolution	33
3.5	Strassen-based Convolution	36
3.6	Summary	38
4	FINE-GRAINED FFT CONVOLUTION	39
4.1	Motivation	39
4.2	A New Data Pattern	41
4.2.1	im2col Process	41
4.2.2	Intra-row Redundancy	42
4.2.3	Inter-row Redundancy	43
4.2.4	Im2col-based Convolution Redundancy	44
4.2.5	Padded Zero Distribution	45
4.2.6	Doubly Block Hankel Matrices	47
4.3	Fine-Grain-FFT-Based Convolution Algorithm	50
4.3.1	Fast Fourier Transform	50
4.3.2	Hankel Matrix Vector Multiplication	52
4.3.3	Hankel Matrices to Circulant Matrices	54
4.3.4	Implicit Element-Wise Matrix Multiplication	55
4.3.5	FFT Hermitian Symmetry	57
4.3.6	Overall Working Flow	58
4.3.7	Arithmetic Complexity Analysis	58
4.3.8	Autotuning	60
4.3.9	Memory Consumption Analysis	60
4.4	Evaluation and Performance Analysis	62
4.4.1	Accuracy	62
4.4.2	Kernel-level Performance	63
4.4.3	Performance in Application	67
4.4.4	Performance Analysis	68
4.5	Summary	69

5	SCALABLE TOP-K QUERY PROCESSING	70
5.1	Overview	70
5.2	Introduction	71
5.3	Related Work	74
5.4	Top-K Query Processing Background	75
5.5	GPU-Based Top-K Query Processing	76
5.5.1	Parallel Index Decompression	77
5.5.1.1	Block Address Calculation.	80
5.5.1.2	Block Decompression.	81
5.5.1.3	Document ID Recovery.	81
5.5.2	Parallel Score Calculation	82
5.5.3	Parallel Top-K Selection	85
5.6	Experiments	87
5.6.1	Comparison with Exhaustive Evaluation (CPU-Based).	88
5.6.2	Comparison with Dynamic Pruning Methods (CPU-Based).	89
5.6.3	Comparison with Previous GPU-Based Method.	91
5.6.4	Time Analysis.	94
5.6.5	Speedup for Different Query Lengths.	98
5.7	Summary	98
6	CONCLUSION AND FUTURE WORK	100
6.1	Conclusion	100
6.2	Future Work	100
	BIBLIOGRAPHY	102

LIST OF TABLES

2.1	Some notations used in the dissertation	13
3.1	Algorithm complexity comparison between direct convolution and FFT-based convolution (See table 2.1 for notations)	31
4.1	Algorithm complexity comparison between FFT-based convolution and our method (See table 2.1 for notations)	58
4.2	Properties of the NVIDIA GeForce GTX TITAN Xp.	62
4.3	FineGrainedFFT convolution absolute element error. Ground truth is computed by im2col+GEMM convolution.	63
4.4	Layer configurations for the three synthetic CNNs. Their performance evaluation is shown in Figure 4.12. Each element in the table indicates (U, K)	67
4.5	Profiling results with varying kernel size. Subscripts of r, f indicate RegularFFT and FineGrainedFFT convolutions. FFT and MM represent the FFTs and element-wise multiplication execution time. (unit ms)	68
5.1	Performance comparison on exhaustive evaluation (ms)	88
5.2	Performance comparison for disjunctive processing (ms): GPU vs. dynamic pruning (K=1,000)	90
5.3	Performance comparison with the GPU baseline in disjunctive mode (ms) (K=2,000)	91
5.4	Performance comparison with the GPU baseline in conjunctive mode (ms) (K=1,000)	91

LIST OF FIGURES

2.1	The LeNet-5 Convolutional Neural Network [47]	9
2.2	The AlexNet Convolutional Neural Network [47]	10
2.3	A 3 by 3 2-D convolution on a 5 by 5 input with no zero-padding and a stride of 1. As the filter slides, it generates a 3 by 3 output based on equation 2.4.	14
2.4	Convolution in CNNs with 5 filters ($f_1 \sim f_5$) and one input. Each filter convolves with the input, and the resultant matrices are stacked to form the output with 5 channels.	16
2.5	Simplified code of convolution computation in CNNs	16
2.6	NCHW VS NHWC data layout comparison example. The red, green and blue elements correspond to R, G, B channel pixels for RGB images.	19
2.7	Comparison of CPU and GPU architecture (Image from Nvidia CUDA C programming guide [58]).	20
2.8	CUDA heterogeneous programming model (Image from Nvidia CUDA C programming guide [58]).	22
3.1	Im2col+GEMM example. A 3 by 3 input is unrolled to a 4 by 4 matrix. It multiplies two 2 by 2 kernels (stored as a 2 by 4 matrix), and generates a 2 by 4 output matrix. Each row corresponds an output feature map.	28
4.1	A 3×3 input with zero-padding of size 1 convolves with a 2×2 kernel (shown in blue on the left). On the right is a larger matrix generated by the im2col process with redundancy.	39
4.2	The close look of im2col process by reshaping the input patch overlapped with a kernel of four rows into a column.	41

4.3	The row kernel slides on the one-row feature map, and it incurs intra-row redundancy. The skew diagonals denoted by the blue arrows are constant, and blue triangles represent zero elements.	42
4.4	As the kernel traverses the entire feature map, the output matrix generated by im2col has inter-row redundancy denoted by dotted blue arrows. Each block is $V \times Q$ with total number of $K \times R$. Refer to table 2.1 for notations.	45
4.5	Three 2D convolution types: valid, same and full convolutions. Each of them is denoted by red, purple and green colors respectively. The input size is 5×5 and the kernel size is 3×3	46
4.6	A new data pattern revealed in figure 4.1 that each block with the same color are the same, and element along the skew diagonals are constant. The elements bounded by red lines correspond to individual rows from the input feature map.	49
4.7	Illustration of radix-2 DIT FFT algorithm	51
4.8	RegularFFT and FineGrainedFFT performance comparison as the kernel size varies from 3 to 8 on synthetic benchmark	63
4.9	RegularFFT and FineGrainedFFT performance comparison as the kernel size varies from 3 to 8 on ILSVRC2017 benchmark	64
4.10	Im2col+GEMM and FineGrainedFFT performance comparison as the kernel size varies from 5 to 45 on synthetic benchmark	65
4.11	RegularFFT and FineGrainedFFT performance comparison as the batch size varies from 1 to 31 while other parameters are fixed.	66
4.12	Layer-wise performance benchmark for the networks composed by five convolutional layers. Others in the legend represents pooling, ReLU and fully connected layers. Conv. layers with maximum speedup are highlighted as shaded rectangles. The average speedup for the three networks is $1.74\times$	67
5.1	An example of index compression	79
5.2	The layout of compressed data for each block in compressed postinglist	80

5.3	An example of two-level search	83
5.4	GPU-OR vs dynamic pruning (TB05)	90
5.5	GPU-AND vs. BL-GPU-AND speed comparison of GPU and BL-GPU as K increases (TB05)	92
5.6	GPU-OR vs. BL-GPU-OR speed comparison of GPU and BL-GPU as K increases (TB05)	92
5.7	GPU-AND vs. BL-GPU-AND speed comparison of GPU and BL-GPU as K increases over three major components (TB05) . . .	94
5.8	GPU-OR vs. BL-GPU-OR speed comparison of GPU and BL-GPU as K increases over three major components (TB05)	95
5.9	Query processing time decomposition for CPU-OR when K=1000 (TB05)	96
5.10	Query processing time decomposition for GPU-OR when K=1000 (TB05)	96
5.11	Speedup over different query lengths (<i>TB05</i>) (K=1000)	97

ABSTRACT

The Convolutional Neural Networks (CNNs) architecture is one of the most widely used deep learning tools. The execution time of CNNs is dominated by the time spent on the convolution steps. Most CNNs implementations adopt a simple yet efficient im2col (image to column) +GEMM approach to implement convolution. The im2col+GEMM approach lowers the convolution into matrix multiplication that can be easily parallelized with highly efficient BLAS libraries. The contribution of this dissertation is that we observe significant but intricately patterned data redundancy in this matrix representation of convolution. We have not been able to identify earlier work that exploits this redundancy to improve the performance of CNNs. In this work, we analyze the origin of the redundancy generated by the im2col process, and reveal a new data pattern to more mathematically concisely describe the matrix representation for convolution. Based on this redundancy-minimized matrix representation, we implement a FFT-based convolution with finer FFT granularity. It achieves on average 23% and maximum 50% speedup on the ILSVRC2017 benchmark over the regular FFT convolution from NVIDIA’s cuDNN library, one of the most widely used CNNs libraries. Moreover, by replacing existing methods with our new convolution method in a popular deep-learning programming framework Caffe, we observe on average 74% speedup for multiple synthetic CNNs in closer-to-real-world application scenarios.

Chapter 1

INTRODUCTION

Deep convolutional neural networks (CNNs) have been very successful in various fields such as image classification in recent years. One of the first successful CNN models can be traced back to [46] and later improved it in [47]. As [42] made significant breakthrough in vision recognition in 2012, more deeper and more complicated network structures are proposed to increase CNN expressiveness [66] [68] [70] [33]. ResNet [33] is 20 times deeper and 8 times deeper than AlexNet [42] and VGGNet [68], respectively. However, it also increases the computational complexity of CNNs, which makes the network more computationally intensive. Furthermore, for some specific real-time interactive applications where the CNN is deployed, and fast response is required, e.g., self-driving cars [7][13], video surveillance systems [23], long latency is not acceptable and fast convolution algorithm is needed. Thus, the computation efficiency of CNN becomes an essential factor in its research and application.

A typical convolutional neural network consists of many different layers. Among these layers, the bulk of the computation is performed on convolutional (CONV) layers [29] [17] [60] [74] [38] [51][61]. It is reported in [38] that the convolution layer accounts for 92% of time distribution of forward pass on a Nvidia K20 GPU for the AlexNet model, using a batch size of 256. The backward pass is similar. [51] also shows that convolutional layer consumes 86%, 89%, 90% and 94% of the total execution time for GoogLeNet, VGG, OverFeat and AlexNet CNN models in one forward and one backward propagation on a single K40c GPU, respectively. Convolution layer requires a substantial amount of computation resources. Especially for modern advanced CNN models with deeper and complicated architecture. Naturally, prior research on

CNNs’ performance has been focused on optimizing the convolution process. Various approaches have been attempted to accelerate it, which we will discuss in chapter 3.

Due to its massive parallel computing capability, GPUs play an important role in the implementation of CNNs. Moreover, Nvidia has developed high efficient drop-in deep learning acceleration library cuDNN [14] with optimized routines on GPUs. Most deep learning frameworks support GPU by default [39] [80] [1] [3] [16]. Implementing a high performance convolution on GPUs is critical to CNN performance.

While a convolution operation can be implemented directly on GPUs, it does not fully utilize the GPU resources efficiently. In order to improve the convolution performance on GPUs, several methods have been proposed [55] [12] [44]. [55] uses FFTs to carry out convolutions in the Fourier domain. Experimental results [74] have shown that larger convolution kernels yield more performance gain. As CovNets utilized smaller kernels [68] [36], Winograd algorithm has been proposed to reduce the amount of multiplication at cost of performing more additions. The savings in multiplication is overwhelmed by the additions for large kernels. On the other hand, the performance of im2col+GEMM approach [12] is consistent with the kernel size since it transforms convolution into matrix multiplication, and then highly tuned GEMM routine can be invoked to compute matrix multiplication. While this method achieves good performance, the redundancy incurred by the im2col operation has been largely overlooked.

We present our work on optimizing CNN’s convolution process at the backend implementation level. Our key insight, and also the main contribution of this work is that we observe significant yet intricate-patterned redundancy hidden in the matrix based representation of CNN’s convolution process. This redundancy has been largely overlooked in prior work, but can be transformed to reduce the computational complexity, and therefore to improve the performance of CNN’s convolution. We present a systematic study of the redundancy and reveal a doubly block Hankel matrix data pattern for an unrolled input feature map. Based on this data pattern, contrary to

the regular FFT convolutions that take 2D FFT over the entire feature map, we implement a new FFT-based convolution with finer granularity, which yields notable performance improvements compared to existing state-of-the-art implementations. We conduct various comparisons, and the experimental results suggest that the fine-grained FFT approach outperforms the regular FFT method for both synthetic and real-world benchmarks.

To reduce the computation complexity and speed up in convolutional layers, many other work have focused on utilizing approximation algorithms [20] [37] or quantizing[30], which create accuracy degradation, we consider them as orthogonal and complementary to our convolution algorithm optimization direction.

We do not consider strided convolutions ($\text{stride} > 1$) throughout this dissertation as our proposed approach is based on the new pattern found on the output matrix generated by `im2col` when stride is one, which we will describe in chapter 4.2. For the related work part, we also do not consider strided convolution in a manner consistent with our new approach. However, strided convolution can be expressed by the sum of multiple convolutions with stride is one by utilizing a reindexing scheme to reorganize the input and filter matrix [9]. In this work, we assume that the feature maps to be square for notational simplicity, but the observation presented generalizes to non-square feature maps with little modification.

The organization of this dissertation is as follows. Chapter 2 provides background on convolutional neural networks(CNNs). We start with the introduction of CNNs, and then describe the core computation of CNNs—convolution. There are several different deep learning frameworks available to build a convolutional neural network, and we chose Caffe as the representative framework to introduce. We also introduce the architecture and programming model of GPUs.

In chapter 3, we review the major existing convolution algorithms in CNNs. Each of them has strengths and weaknesses in computing convolution, and excels at different parts of parameter space in CNNs. In the next chapter, we propose our approach to efficiently compute convolution in certain parts of the parameter space.

Chapter 4 describes our proposed fine-grained FFT convolution, a new computationally efficient convolution that is distinguished from existing convolution approaches. The algorithm is based on a new data pattern that is revealed in the im2col process. We analyze the data pattern in a bottom-up manner. It begins with intra-row redundancy and inter-row redundancy, and integrates them into im2col-based convolution redundancy. A connection between this redundancy and the data pattern of doubly block hankel matrices is built. Based on the data pattern, we propose our fine-grained FFT convolution. Its detailed implementation as well as arithmetic and memory analyses are presented. We conclude this chapter with evaluation and performance analysis.

We also present the other work which utilizes graphics processing units (GPUs) to optimize topK query processing in chapter 5. With abundant parallelism provided by GPUs, our proposed framework is scalable with respect to K (the number of returned documents). We optimize the three main steps of topK query processing, namely index decompression, score calculation and topK selection, using GPUs. We test our framework against previous state-of-the-art CPU and GPU topK query processing approaches, and the proposed approach is more efficient and scalable.

In chapter 6, we summarize our contributions and conclude our work, and consider the natural steps for our further work.

Chapter 2

BACKGROUND

Understanding the architecture of convolutional neural network (CNN) and its convolution operation is a key aspect to the convolution optimization. In this chapter, we present an overview of CNN architecture and its main types of layers. Specifically, we focus on the convolutional layer. All these layers are stacked to form a CNN model. We introduce two representative CNN models, and how to train them and use the trained models for inference. Then we discuss the core convolution operation CNNs perform. We also present a brief introduction of Caffe deep learning framework to which our optimized convolution algorithm is integrated. To obtain high performance, the framework mainly runs on GPUs. We present the GPU architecture and programming model. We conclude this chapter with circulant matrices, a mathematical tool that will be used in our proposed FFT convolution.

2.1 Convolutional Neural Network

In this section, we describe the major components of a convolutional neural network. We start with the overview of CNN architecture, followed by the description of three common types of layers, namely convolutional, pooling, and fully-connected layers. We use two representative models Lenet5 and AlexNet as examples to demonstrate how these layers are connected to form a CNN. Then we briefly discuss training and inference.

2.1.1 Convolutional Neural Network Architecture

The architecture of a typical CNN is composed of multiple stages [45]. Each stage consists of a convolutional layer, a non-linearity layer and a pooling layer, where

non-linearity is introduced into CNNs using non-linearity layers and pooling layers help the output feature map to be robust and invariant to small shifts and distortions in the previous layer [49]. The last layer of CNNs is a fully-connected (FC) layer which combines the results of convolutions of a set of relatively high-level features for classification purpose.

2.1.2 Convolutional Layers

Convolutional Networks (ConvNets) consists of multiple convolutional layers to extract features from the input. Convolutional layers first perform convolution operations which will be discussed in detail in section 2.2. They also add a bias term, and then apply a non-linear activation function such as rectified linear unit (ReLU). It is typically applied to the convolutional layer to introduce non-linearity and better approximate non-linear features of the input data. Mathematically, we have that

$$y_j = f \left(\sum_i x_i * k_{ij} + b_j \right) \quad (2.1)$$

where x_i and y_j denote input feature maps and output feature maps, respectively. k_{ij} is the convolution kernel, and b_j is the additive bias. $f(\cdot)$ and $*$ represent the non-linear activation function and convolution. For a convolution layer that transforms an input image with c channels into an output image with c' channels, the total number of convolutions is $b \cdot c \cdot c'$ if the batch size is b .

Convolutional layers act as a feature extractor that extract feature maps by different convolutional kernels. Each kernel extracts a feature from input image, and a convolutional layer typically uses multiple kernels to extract multiple feature maps. These features are learning automatically by stacks of convolutional layers with powerful representation capability, not designed by human engineers [45]. Low-level feature extracted at lower convolutional layers are combined to more abstract features at higher layers. In convolutional network, the input and output to the convolutional layer is often referred to as feature maps. Each neuron at the current convolutional layer is connected to a local region of the previous layer with the full depth dimension, which is

referred as local receptive field. It results in a reduced number of connections between layers by only connecting to the local receptive field of previous layers. The property associated with the sparsely connected layers is called sparse connectivity [25]. Compared with fully connected layers that each neuron is connected to every neuron in the previous layer, it drastically reduces the number of parameters and computational complexity of the network. Another key idea of CNNs is parameter sharing [25]. It is accomplished by applying the same weights over the input feature maps at all positions to extract the output feature maps. It further reduces the storage requirement for the parameters. The output feature maps represent a particular feature extracted from the input. Convolutional layer produces a feature map for each filter, thus the number of output feature maps depends on the number of filters. The feature map is usually a tensor of data and the kernel is also a tensor of parameters. In CNNs, tensors are nothing more than n-dimensional arrays, e.g., one dimensional tensor is a vector and two dimensional tensor is a matrix.

The convolutional layer consists of two types of parameters: learnable parameters and hyperparameters. The learnable parameters kernel k and bias b in equation 2.1 are learned by backpropagation. The total number of learnable parameters for the convolutional layer is $(U \times V \times C + 1) \times K$, added 1 because of the bias term for each kernel. On the other hand, hyperparameters are set before training and can not be learned. They determine the convolutional layer structure. The receptive field is a hyperparameter which defines the connectivity of each neuron in the current convolutional layer to the previous layer. It is equivalent to the kernel size. The other three hyperparameters control the output of the convolutional layer: the number of kernels, which also corresponds to the output channels; stride and padding. Stride controls how the kernel convolves around the input feature map, and padding specifies the amount of zeros padded around the border.

2.1.3 Sub-sampling and Fully-connected Layers

In addition to convolutional layers, CNNs also contain sub-sampling and fully-connected layers. Sub-sampling layers are also known as pooling layers, which it partitions the input into patches and output a value based on the non-linearity function, e.g., each patch outputs the maximum in max-pooling. Formally,

$$y_j = d(x_j)$$

where $d(\cdot)$ represents a sub-sampling function. Max-pooling and average pooling are two common operations used in this layer. Pooling layers are often used after convolutional layers to reduce the dimension of feature maps. Once the feature is detected, its exact location is not important and can be simplified in order to provide some degree of shift and distortion invariance. In contrast to convolutional layers, pooling layers has no trainable parameters. In LeNet5, it performs average pooling by computing the average of a 2×2 area in the feature map of previous layer and the condensed feature map has half the size of the previous one. The last few layers in CNNs are fully-connected layers. They combine the results of convolutions of a set of relatively high-level features and merge features distributed at different locations for classification purpose, whereas convolutional layers extract the feature.

Each neuron in the convolutional layer is only connected to a local region of previous layer with the same set of weights. In contrast, each neuron in the fully-connected layer is connected to every neuron in the previous layer, and every connection has different weights. As a result, fully-connected layers contribute to the majority of parameters in comparison to the other layers in CNNs. Computations in fully-connected layers are carried out as standard matrix multiplications. The output is fed into a softmax layer which acts as classifiers that output a N dimensional vector, where N is the number of classes from which the input can choose. Each number in the N dimensional vector represents the probability of the input being associated with the class in the softmax layer.

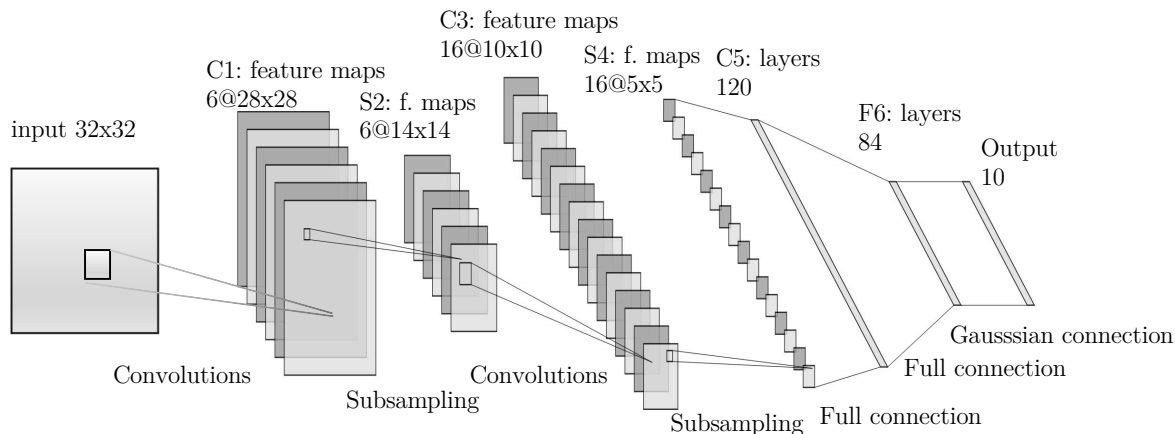


Figure 2.1: The LeNet-5 Convolutional Neural Network [47]

2.1.4 Representative CNN Models

In order to better understand CNNs, we choose two representative CNN models, LeNet-5 and AlexNet, to illustrate the architecture of CNN models. LeNet-5 shown in figure 2.1 is the first successful convolutional neural network designed to identify hand-written digits in the MNIST dataset [48]. Although it is simple, it has all the aforementioned layers to connect as a fully functional CNN. LeNet5 composes of two sets of convolutional and subsampling layers, followed by two fully connected layers and one final fully connected softmax layer. Note that layer C5 is a convolutional layer with $120 \times 1 \times 1$ feature maps. Each of the neurons in C5 is connected to the previous layer S4 with kernel size 5×5 . Because the size of feature maps in S4 is also 5×5 , it means that C5 is fully connected to S4 and it can be regarded as a fully-connected layer. The subsampling layer uses a form of average pooling. The input to LeNet5 is a 32×32 pixel grayscale image and it produces a vector with 10 values representing the probability of digits from 0 to 9. Thus, a handwritten character can be recognized.

In 2012, AlexNet [42] developed by a group from the University of Toronto won the 2012 ImageNet LSVRC-2012 competition, and dropped the error rate significantly and showed groundbreaking results. As the first successful modern deep neural network, it added a huge boost to the deep neural network approach. AlexNet has a very

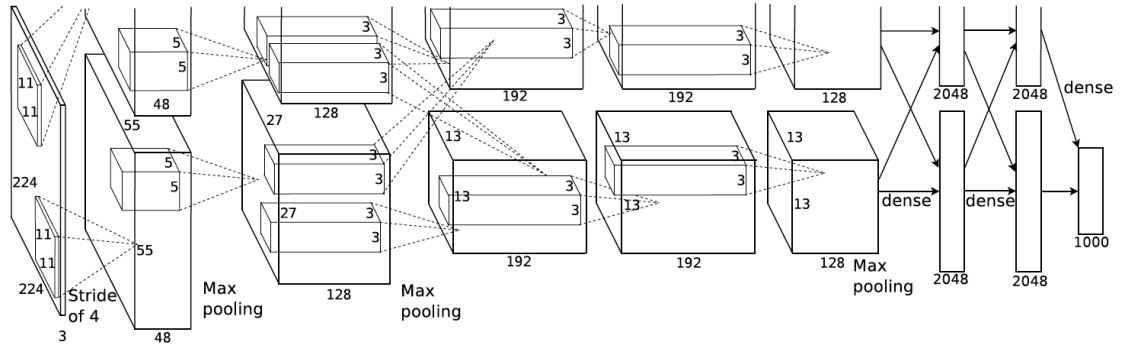


Figure 2.2: The AlexNet Convolutional Neural Network [47]

similar architecture as LeNet-5 but one major difference is that AlexNet has a deeper network. As is shown in figure 2.2, it has 5 convolutional layers and 3 fully connected layers. Only the first two and the fifth convolutional layers are followed by max-pooling layers, which select the maximum value in the pooling window. In contrast, the average pooling layer is used in LeNet-5. The output of the fifth convolutional layer is then fed into a series of two fully-connected layers. The last layer in AlexNet is a softmax classifier with 1000 class labels. In addition, Rectified Linear Unit (ReLU) non-linearity is applied after every convolutional and fully connected layer. As we can see from the figure, the network is split into two pipelines on two GPUs mainly because the GPU memory size is limited and the network training is too big to fit on one GPU. With the success of AlexNet, it inspired more deeper neural networks that achieves stunning results in computer vision.

As we can see from Lenet-5 and AlexNet, a typical structure of convolutional neural networks consists of multiple convolutional layers. The amount can be up to a thousand layers [33] in modern CNN models. With the deep architecture, modern CNNs are able to achieve high performance. Non-linearity and pooling layers are applied after convolutional layers, and lastly a small number of fully-connected layers

are applied at the end of CNN. Beginning from 2012 that AlexNet achieved state-of-the-art recognition accuracy in image classification, there are many different modifications to the structure of CNNs such as GoogLeNet, VGG-16 and ResNet etc.. The ResNet even reached human-level accuracy in computer vision tasks.

2.1.5 Training and Inference

The learning of CNNs referred as training is a process the networks learn from input data. Once the output is produced, a loss function is invoked to measure the distance between output values and actual values. The gradient of loss is the partial derivative of loss function with respect to weights, which indicates how to change weights in order to reduce the loss function. The gradient is back propagated to previous layers, and parameters are updated by gradient descent algorithm to get better feature representations. The algorithm continuously minimizes the cost function of CNN model on the training dataset until the network is converged. It usually consists of tens to hundreds of epochs, where one epoch constitutes of one forward pass and one backward pass of the entire training dataset and it is divided into small mini-batches. In practice, CNNs usually operate in minibatches, because the stochastic gradient descent [63] that is used to update the weights works well with minibatch. In addition, it amortizes GPU memory cost when loading multiple inputs and kernels in mini-batch mode. Batch size is a convolutional parameter that has performance impact on convolutions. In the forward pass, each batch traverses the network and each layer applies its transformation to the batch and produces the output. The other pass is backward pass where each layer back-propagates the gradients to earlier layers in the backward direction. Once the network is trained and converged, we run inference using the weights learnt at the training phase. Inference applies the trained CNN to machine learning tasks to get an inferred output. Unlike training, it does not have backward pass.

2.2 Convolution in CNNs

Our goal is to achieve high efficiency on convolution operations in CNNs, and it is desired to fully study and understand this operation. We start with a high-level overview of the convolution operation. In particular, we consider 2-d discrete convolution in image processing. We also summarize some notations used throughout this work in table 2.1. We then move on to the batched convolution that computes the convolution of batches of images and filters in CNNs. For clarity, we also fix the terms used with similar meaning in the context of CNNs to clear up ambiguity. We provide the seven-nested loops implementation in figure 2.5 and analyze the theoretical algorithmic complexity of convolution. We conclude this section with convolution in backward pass.

2.2.1 2D Convolution

A convolution (cross correlation) takes two functions as inputs and measure the similarity between the two. For discrete functions, it is defined as follows:

$$(f * g)[n] = \sum_{m=0}^{M-1} f[n+m]g[m] \quad (2.2)$$

Where M, N denotes the length for two finite sequences f and g, respectively. It can be naturally extended to higher dimension. To measure the similarity in spatial domain, we apply two-dimension between image(I: $0 \leq h \leq H-1, 0 \leq w \leq W-1$) and filter(F: $0 \leq u \leq U-1, 0 \leq v \leq V-1$), which is defined in the following equation:

$$conv_{2D}(I, F)[h][w] = \sum_{u=0}^{U-1} \sum_{v=0}^{V-1} I[h+u][w+v]F[u][v] \quad (2.3)$$

Essentially, convolution operation performs the dot product between the filter and the corresponding values in the image. Filters, also known as convolutional kernels, and they are used interchangeably in this dissertation, however, it should not be confused with GPU kernels, which are functions executed on GPUs. It is worth pointing out that equation 2.2 is different from the mathematical definition of convolution

Table 2.1: Some notations used in the dissertation

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

since there is no flip of filter as it slides across the image, so it should be called cross-correlation. However, this difference does not affect the performance of the operation in CNNs, it is only one convention. Flipping the kernel will make CNNs learn the flipped version of the learned kernel, thus CNNs implement cross-correlation but call it convolution.

2.2.2 CNN Convolution

Table 2.1 establishes some notations for the material presented in the dissertation. In figure 2.3, the first element of the output is produced by the dot-product of kernel by a sub-matrix of input (the blue region). As the kernel slides over the input in both horizontal and vertical directions, each subsequent element is generated. The output width is determined by the following formula.

$$Q = (W - V + 2P)/S + 1 \tag{2.4}$$

Where Q , W , V , P , and S are explained in table 2.1. The latter three hyperparameters control the output width. CNNs perform sliding window-style convolutions, and the kernel slides from the leftmost part of input feature map to the right side. The output width is defined as the number of placements of kernel on the input in equation 2.4, where plus one accounts for the initial placement of the kernel. We only consider the

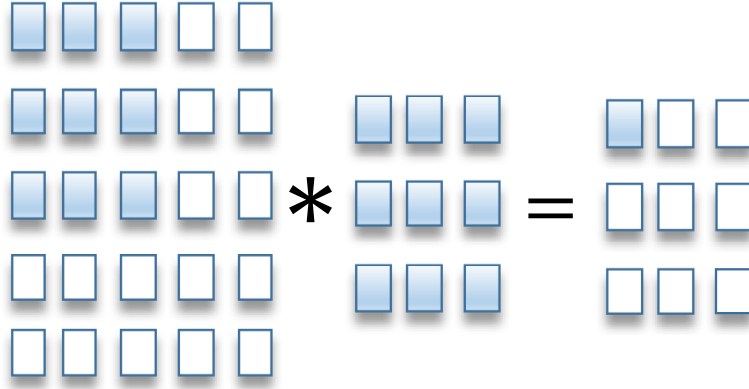


Figure 2.3: A 3 by 3 2-D convolution on a 5 by 5 input with no zero-padding and a stride of 1. As the filter slides, it generates a 3 by 3 output based on equation 2.4.

width here, but it can be trivially applied to the height. In figure 2.3, the output image have the same height and width size. Above analysis assumes images have only one channel, which we only consider spatial dimensions (height and width) ignoring the depth dimension. In CNNs convolution operations apply multiple filters over images with multiple channels in a mini-batch. Channels for images are also known as feature maps in CNNs, and they can be used interchangeably in this work. Let us formally define the convolution in CNNs. For a given convolutional layer, we have inputs x with a batch size N , and each of them has a set of feature maps C , thus each input feature map is represented as $x_{(n,i)}$, $n \in N, i \in C$. They are convolved $K \times C$ different kernels $f_{(j,i)}$, $j \in K, i \in C$, and each output feature map is $y_{(n,j)}$. CNNs convolution can be formulated as follows:

$$y_{(n,j)} = \sum_{i \in C} conv_{2D}(x_{(n,i)}, f_{(j,i)}) \quad (2.5)$$

To convolve the multiple channels with the filter, CNNs perform a 2-d convolution separately in corresponding channels and sum results across all the channels. Convolution in CNNs requires that the filter must have the same number of channels as the input. As the convolution with batch size of 1 illustrated in figure 2.4, the first output channel is the result of convolutions of corresponding channels of filter f_1 and the input. For multiple filters, the output can be viewed as the concatenation of the resultant matrices generated by the input and corresponding filters. Until now,

we have shown how CNNs convolution is extended from 1-d convolution. To further analyze its computation, the seven-nested loops pseudocode for the convolution is illustrated in figure 2.5. It is also the pseudocode for direct convolution, which will show in subsection 3.1, but it is a parallel GPU implementation. The innermost three accumulation loops compute the actual convolution, which are dependent and can not be parallelized, whereas the remaining loops do not have dependency. The innermost two loops perform the computation described in 2.3, whose algorithmic complexity is $U \cdot V$ FLOPs. The remaining loops has $N \cdot K \cdot R \cdot Q \cdot C$ iterations. Therefore the total complexity is $N \cdot K \cdot R \cdot Q \cdot C \cdot U \cdot V$ FLOPs, where a single multiply-accumulate operation counts as one operation.

In the previous section 2.1.2, we have discussed that the convolutional layer has four hyperparameters (parameters), kernel size (kernel height and kernel width), number of kernels, stride, padding. The convolution parameter in CNNs include four additional parameters with regard to the input feature maps: input channels, which also correspond to the number of input feature maps; batch size, input height and input width. The convolution parameters are essentially the sum of convolutional layer parameters and input parameters. If stride and padding have different values along height and width, both of them have different parameters along height and width, S_h , S_w , P_h , and P_w . In total, the convolution parameter space could be 11-dimensional.

2.2.3 Convolution in Backward Pass

In forward propagation, the input image is fed to input layer, and pass through a stack of convolutional layers which the input feature maps are convolved with different kernels to produce output feature maps. When the input data has passed all of layers, a loss function is used to compare the CNN results with correct results to calculate errors. During the backward pass, the gradient is propagated to previous layers to update the parameters in kernels to minimize the cost function using stochastic gradient descent. A key property of CNN is that gradient with respect to input layers can be computed

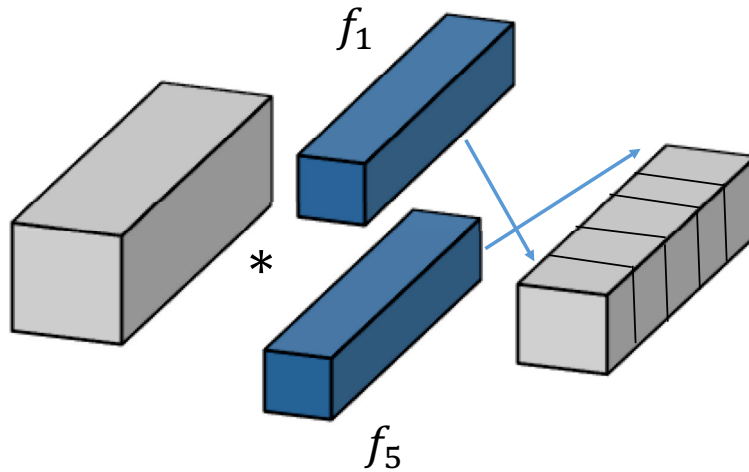


Figure 2.4: Convolution in CNNs with 5 filters ($f_1 \sim f_5$) and one input. Each filter convolves with the input, and the resultant matrices are stacked to form the output with 5 channels.

```

1  Input[N] [C] [H] [W];
2  Filter[K] [C] [U] [V];
3  Output[N] [K] [R] [Q];
4  for(n = 0; n < N; n++)
5    for(k = 0; k < K; k++)
6      for(r = 0; r < R; r++)
7        for(q = 0; q < Q; q++)
8          float val = 0;
9          for(c = 0; c < C; c++)
10             for(u = 0; u < U; u++)
11                for(v = 0; v < V; v++)
12                   val += Filter[k, c, u, v] * Input[n, c, r + u, q + v];
13             Output[n, k, r, q] = val;

```

Figure 2.5: Simplified code of convolution computation in CNNs

from the gradient with respect to output layers, consequently, the gradient with respect to the weight can be computed from these values. Since CNNs typically use stochastic gradient descent technique [63], the gradient of loss with respect to weights is accumulated across different images in the same batch. This process repeats until each layer updates their weights and one iteration of forward and backward pass finishes. Both the gradient backpropagation and gradient accumulation involve convolution operation in the backward pass. Similar to equation 2.5, we formally define them as follows:

$$\frac{\partial L}{\partial x_{(n,i)}} = \sum_{j \in K} conv_{2D}\left(\frac{\partial L}{\partial y_{(n,j)}}, f_{(j,i)}\right) \quad (2.6)$$

Equation 2.6 defines the gradient backpropagation which the gradient of loss function with inputs are computed by convolving the gradients with respect to outputs with the kernels. Thus, the gradients can be propagated backwards to previous layers. Note that $conv_{2D}$ here is a full convolution and the kernel needs to be rotated by 180 degree.

$$\frac{\partial L}{\partial f_{(j,i)}} = \sum_{n \in N} conv_{2D}\left(\frac{\partial L}{\partial y_{(n,j)}}, x_{(n,i)}\right) \quad (2.7)$$

The gradients with respect to the kernels are computed by the convolutions between the inputs and the gradients with respect to the outputs. Note that all convolutions in equations 2.5 2.6 2.7 is 2D convolutions between sets of 2D matrices.

2.3 Deep Learning Frameworks

Many deep learning frameworks have emerged to build efficient CNN models, such as Caffe [39], CNTK [80], TensorFlow [1], Theano [3], and Torch [16]. We choose a representative framework, Caffe[39], as the underlying framework to compare our proposed convolution method with other existing methods. Thus we briefly describe its main highlights and overall architecture. We also compare two major data layout used in the deep learning frameworks. Lastly, we introduce ImageNet benchmark, which is one of the most widely used benchmarks in deep learning for performance evaluation.

2.3.1 Caffe

Caffe [39] is an open source framework developed by UC Berkeley in 2014, which was initially designed for computer vision applications, and it was soon adopted by users from other application domains because of the expression, speed, and modularity nature of Caffe. Due to its expressive architecture, CNN models are defined by configuration files using the Protocol Buffer language. It is simple to build a CNN architecture and define an optimization as configuration files. Caffe provides a complete list of layer types for a CNN model, such as convolution, pooling, etc. Each type of layers has a generic interface. It allows users to define complicated networks by composing these layers. The first layer is a data layer that feed inputs to the network. Each subsequent layer performs a data transformation based on its specific configuration file. The last layer corresponds to a loss layer. Caffe encapsulates low-level implementation details in layers, and communicates by blobs between layers. Blobs are simply 4-dimensional arrays to store data, gradients, and model parameters. The conventional blob dimensions in Caffe adopt NCHW data layout, which we will discuss in the following section. The unified memory interface handles memory allocation and synchronization between CPU and GPU efficiently. To efficiently process memory operations, it allocate memory on CPU and GPU on demand and synchronize between CPU and GPU as needed. In the forward pass, layers takes input data blobs and produce output data blobs, while gradient blobs are back-propagated to previous layers during the backward pass.

2.3.2 Data Layout

The data structures used by convolution layers are typically four dimensional tensors and there are two most popular data layouts NCHW and NHWC (see table 2.1 for notations). The stride in memory layout is ordered from small to large, where consecutive element in width dimension is contiguous in memory and dimension N has a stride of $C \times H \times W$ for NCHW and NHWC. In comparison, the elements along the lowest dimension C are stored contiguously in NHWC layout. Figure 2.6 illustrates a comparison example between NCHW and NHWC in the channel (C) dimension. Elements

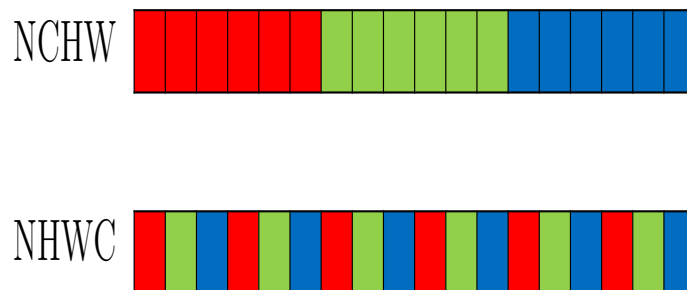


Figure 2.6: NCHW VS NHWC data layout comparison example. The red, green and blue elements correspond to R, G, B channel pixels for RGB images.

from the same channel are adjacent in memory for the NCHW format, whereas elements from different channels are adjacent in memory for the NHWC format¹. The default layout in Caffe is NCHW. Other frameworks CNTK[80], Theano[3], and Torch[16] also choose NCHW as the default tensor format. However, NHWC is the TensorFlow[1] default and NCHW is the optimal format when using GPUs. If Tensorflow uses algorithms when the input is in the format of NHWC, it internally converts it to NCHW. It converts back to NHWC when the algorithm finishes. The conversion incurs overhead and [40] shows that changing the tensor format from NHWC to NCHW leads to about 15% performance improvement when training AlexNet for TensorFlow. Caffe employs the NCHW data layout and we take NCHW as our default data layout in this work.

2.3.3 ImageNet Benchmark

ImageNet [19] is one of the most popular datasets in deep learning, which contains more than 14 million images and 20,000 classes. These classes is built upon the WordNet structure to organize images. An excellent example of the successes of ImageNet dataset can be illustrated with the milestone AlexNet CNN model in 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC). The annual competition uses subsets of the ImageNet dataset to benchmark CNN models. In the object localization task of ILSVRC2017, the dataset contains 1.3M training images, 100,000

testing images and 50,000 validation images. We use the validation dataset to benchmark our proposed convolution algorithm.

2.4 Graphics Processing Units (GPUs)

Unlike central processor units (CPUs), Graphics Processing Units (GPUs) devote more transistors to arithmetic logic units rather than caches and flow control logic, as depicted in Figure 2.7. CPUs use memory caching and sophisticated flow control to avoid stalls and gain efficiency. In contrast, GPUs offer a large number of ALUs to perform operations in parallel in a SIMD (single instruction multiple data) fashion to gain high arithmetic throughput. They use massive parallelism to hide memory access latency instead of caches for CPUs. Because GPUs are throughput-oriented systems, they are suited for applications with massive parallelism.

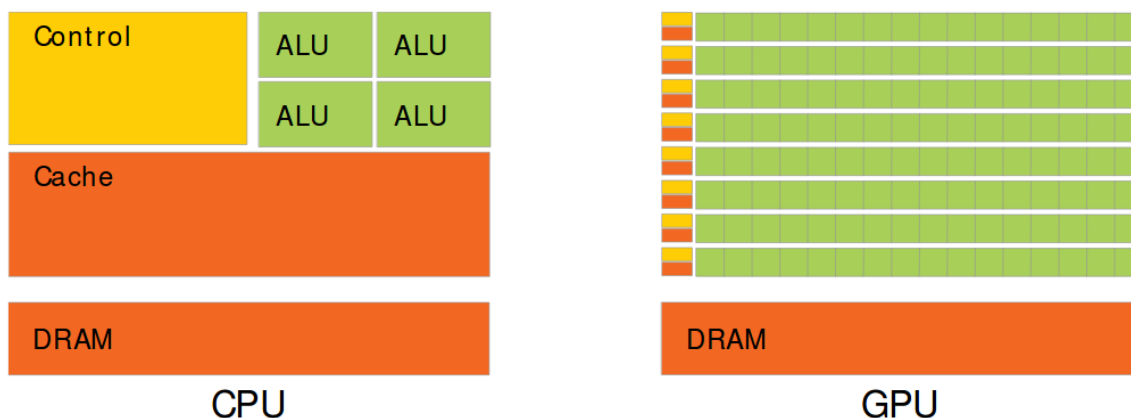


Figure 2.7: Comparison of CPU and GPU architecture (Image from Nvidia CUDA C programming guide [58]).

The massive parallel computing capability of GPUs is an ideal platform for CNNs with substantial compute and memory operations requirement. With the advent of GPGPU, which is referred to as general-purpose computing on GPUs, they are suited for high compute- and data-intensive operations in CNNs. In addition, the deep learning library cuDNN [14] with several optimized convolution algorithms used as backend in Caffe makes GPUs more efficient in the applications for CNNs. One

programming platform widely used in GPGPU community is Compute Unified Device Architecture (CUDA) [24]. Figure 2.8 shows the CUDA programming model. The serial program executes on the host (CPU) while parallel kernel runs on the device (GPU). Both CPU and GPU have separate memory space connected through PCI Express. The GPU kernels are launched asynchronously from the point of view of the CPU, and they are functions run on GPU. GPU programming model has a multi-level thread hierarchy that kernels are executed by a grid of thread blocks and each thread block is composed by a batch of threads. CUDA also has a multi-level memory hierarchy, which consists of global memory, per-block shared memory, and per-thread local memory. Threads in a block are organized as groups of 32 threads (warps) performing essentially vector operations, and they can communicate with each other by the fast shared memory. Each thread and thread block have a specific ID, and GPU can run thousands of threads in parallel.

Particularly, a direct convolution of an image $H \times W$ with a kernel $U \times V$ has computational complexity of $\mathcal{O}(H \cdot W \cdot U \cdot V)$, and total data movement of $\mathcal{O}(H \cdot W)$. Therefore, convolutions are compute-bound theoretically. With the abundant parallelism provided, the convolution operation performance would be boosted if the parallelism is fully exploited.

2.5 Circulant Matrices

In this section, we introduce circulant matrix, which plays an important role in our proposed convolution. We also present a derivation of diagonalization of circulant matrices by DFT matrices. It further turns out circulant matrix multiplication can be carried out by efficient FFTs. This is the basis for our convolution optimization.

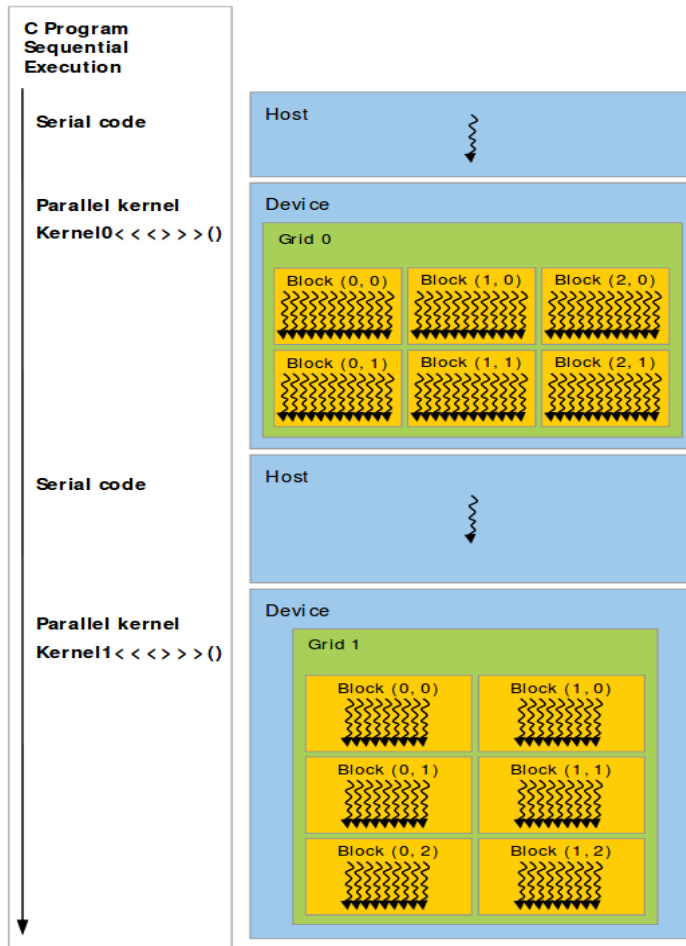


Figure 2.8: CUDA heterogeneous programming model (Image from Nvidia CUDA C programming guide [58]).

A $n \times n$ circulant matrix has the following form.

$$X = C(\mathbf{x}) = \begin{bmatrix} x_1 & x_2 & x_3 & \cdots & x_n \\ x_n & x_1 & x_2 & \cdots & x_{n-1} \\ x_{n-1} & x_n & x_1 & \cdots & x_{n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_2 & x_3 & x_4 & \cdots & x_1 \end{bmatrix}$$

It is completely specified by the first row, which is also known as the generating vector \mathbf{x} . Each subsequent row is obtained by doing a right-shift of the previous row by one, wrapping around at the edges. Each entry of X_{kj} is given by $X_{kj} = x_{(j-k) \bmod n}$.

According to the definition of eigenvalues and eigenvectors, eigenvalues λ_k and eigenvectors v_k of the circulant matrix X are defined by

$$Xv = \lambda v$$

It is equivalent to n functions

$$\sum_{k=0}^{m-1} x_{n-m+k} v_k + \sum_{k=m}^{n-1} x_{k-m} v_k = \lambda v_m; m = 0, 1, \dots, n-1$$

The above functions can be written as follows if we change the summation variables

$$\sum_{k=0}^{n-1-m} x_k v_{k+m} + \sum_{k=n-m}^{n-1} x_k v_{k-(n-m)} = \lambda v_m$$

we choose $v_k = \psi^k$

$$\sum_{k=0}^{n-1-m} x_k \psi^k + \psi^{-n} \sum_{k=n-m}^{n-1} x_k \psi^k = \lambda$$

In addition, we also choose $\psi^{-n} = 1$, which means ψ is a root of unity. Thus we have an eigenvalue for X

$$\lambda = \sum_{k=0}^{n-1} x_k \psi^k$$

and its corresponding eigenvector

$$v = n^{-1/2} (1, \psi, \psi^2, \dots, \psi^{n-1})'$$

where prime denotes transpose. $n^{-1/2}$ is the length of the eigenvector, and it is used to normalize the eigenvector. Because ψ is the root of unity and $\psi_m = e^{-2\pi im/n}$, the eigenvalue becomes

$$\lambda_m = \sum_{k=0}^{n-1} x_k e^{-2\pi imk/n}$$

It is in fact the discrete Fourier transform (DFT) of the sequence $\{x_k\}$. The corresponding eigenvector is

$$v_m = \frac{1}{\sqrt{n}} (1, e^{-2\pi im/n}, \dots, e^{-2\pi im(n-1)/n})'$$

The definition of eigenvalues and eigenvectors for X becomes

$$Xv_m = \lambda_m v_m, m = 0, 1, \dots, n-1$$

The above equation can be written as a single equation

$$X\mathbb{F} = \mathbb{F}\Delta$$

where columns in the matrix \mathbb{F} are the eigenvectors. Eigenvectors for circulant matrices are the same and \mathbb{F} is a DFT matrix.

$$\mathbb{F} = (v^{(0)} \ v^{(1)} \ \dots \ v^{(n-1)})$$

Thus we have

$$X = \mathbb{F}\Delta\mathbb{F}^{-1}$$

where Δ is a diagonal matrix containing the eigenvalues of X , $\lambda_0, \lambda_1, \dots, \lambda_{n-1}$, such that $\Delta = \text{diag}(\mathbb{F}x)$. It is diagonalized by the Discrete Fourier Transform (DFT) matrix regardless of vector x [26].

Therefore, the fast circulant matrix-vector product is as follows.

$$Xy = \mathbb{F}\Delta\mathbb{F}^{-1}y = \mathbb{F}(\mathbb{F}x \circ \mathbb{F}^{-1}y)$$

where y is n -dimensional vector and \circ denotes the Hadamard element-wise vector multiplication.

It first computes a DFT $\mathbb{F}x$ and an IDFT $\mathbb{F}^{-1}y$ and then a final DFT $\mathbb{F}(\mathbb{F}x \circ \mathbb{F}^{-1}y)$. By applying FFTs, these DFTs can be computed in $O(n \log n)$ operations.

Let the second matrix Y compose of m n -dimensional vector such that $Y = [y_1, y_2, \dots, y_m]$. Similarly, the circulant matrix-matrix multiplication is shown below.

$$XY = \mathbb{F}\Delta\mathbb{F}^{-1}Y = \mathbb{F}(\mathbb{F}x \circ \mathbb{F}^{-1}[y_1, y_2, \dots, y_m])$$

The multiplication can be computed by taking the inverse FFT of each column in matrix Y , and do the point-wise multiplication with the Fourier transform of generating vector x of X , followed by the final FFT. Thus for the circulant matrix X with dimension $n \times n$ and another matrix Y with dimension $n \times m$, the computational complexity is $O(mn \log n)$ as opposed to $O(mn^2)$ for naive matrix multiplication.

Chapter 3

EXISTING CONVOLUTION ALGORITHMS

Convolution has been studied extensively by the research community and the industry since convolutional neural network(CNN) has achieved great successes in a large number of applications in recent years. Due to its success, many efficient convolution methods have been proposed. We begin with describing the direct convolution approach. Despite its simplicity, it is shown significant performance in a seminal paper by Krizhevsky et al. [42]. We then describe four indirect approaches: im2col-based convolution, FFT-based convolution, Winograd convolution and Strassen-based convolution. In fact, it is worth noting that there is no single convolution method that performs best for all the scenarios in CNNs. The state-of-the-art cuDNN [14] library as the backend for all major deep learning frameworks includes most of the following convolution approaches and it runs all of internal approaches to determine which algorithm offers the best performance for a given problem size and memory constraints. To further understand the landscape of this research area, we identify strengths and weaknesses of convolution methods, which could potentially guide our proposed convolution.

3.1 Direct Convolution

Direct convolution is a straightforward way to perform convolution. As illustrated in in algorithm 2.5, the pseudocode represents how direct convolution works in serially. Direct convolution on GPUs is nothing more than implementing the for-loop in parallel. For-loops in lines 4-7 in algorithm 2.5 are independent and can be parallelized, while for-loops in line 9-11 are not independent because a shared data dependency occurs and all of the loops are accumulated into the same val variable. One

typical implementation is that each element in the output feature map is computed by one thread, and each output feature map is computed by a thread block since threads in the same block can communicate with each other by faster on-chip shared memory. Thus it will not expose sufficient parallelism to fully utilize the resources on GPUs. Compared with other convolution approaches, the advantage of the direct convolution is clear: instead of utilizing temporary memory to keep intermediate data, it does not need additional memory because it computes the convolution directly. Cuda-convnet2 [41] is one of the earliest CNN frameworks with direct convolution implementation. It achieves high efficiency when batch size is large [14], but it generalizes poorly once batch size is 64 or below. Additionally, it supports limited configuration shapes, e.g., images and filters must be square, and the filter number must be a multiple of 16 [51]. Lavin et al. [43] propose an efficient convolution maxDNN based on SGEMM implementations created by an open source assembler for NVIDIA Maxwell GPUs [27], which is developed by reverse engineering the binary of Maxwell kernels. In maxDNN, a single output feature map coordinate is computed by a matrix multiply between unrolled input matrix with size $N \times UVC$ and the filter matrix of size $UVC \times K$ (see table 2.1 for notations). maxDNN is written in low-level Maxwell assembler, and it takes advantage of hardware specific optimizations and obtains high performance, however it is not portable and the high efficiency convolution is not accessible for other architectures.

3.2 Im2col+GEMM Convolution

Im2col+GEMM, also known as lowering or unrolling convolution, which is a straightforward and efficient approach to compute convolution. Im2col (image to column) is well known that image patches based on the kernel size are rearranged into columns and reorganized them into a concatenated matrix. Im2col-based convolution first unrolls/lowers input images to 2d matrices, and kernels are already stored as the kernel matrix, thus convolution is converted to a general matrix multiply (GEMM). Each row of the output matrix corresponds to an output feature map. To make the

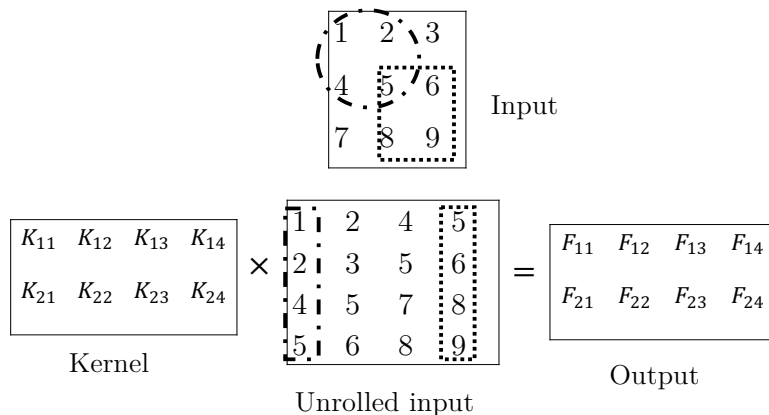


Figure 3.1: Im2col+GEMM example. A 3 by 3 input is unrolled to a 4 by 4 matrix. It multiplies two 2 by 2 kernels (stored as a 2 by 4 matrix), and generates a 2 by 4 output matrix. Each row corresponds an output feature map.

discussion more concrete, we present a simple example to illustrate how this approach works. In figure 3.1, for simplicity, we have one 2D input and two kernels with size 3×3 and 2×2 , respectively, omitting the depth dimension. The first image patch indicated by the dotted circle is reorganized into the first column in the unrolled input. Similarly, the last patch (5, 6, 8, 9) is the last column in the unrolled input. By multiplying the unrolled input with the kernel matrix, each row of the output matrix contains the convolution between the input and corresponding kernel. In CNNs, the kernel matrix has dimensions $K \times CUV$, and the input data with dimension $NCHW$ is unrolled to N matrices with dimensions $CUV \times NRQ$ (see table 2.1 for notations). Note that each element in the original input is duplicated up to UV times. It is obviously that large kernel size has more duplication and requires large temporary memory to store the unrolled input.

The kernel matrix multiplies the unrolled input matrix resulting in the convoluted output feature. The central idea of this approach is to unroll and duplicate the input such that the convolution problem is transformed to a well-studied matrix multiplication problem. Therefore, it is strongly dependent on the performance of GEMM implementation. The performance of matrix multiplication routine in cuBLAS does not change linearly even for a slight change in matrix dimensions [59]. It is due to the

fact that cuBLAS chooses one out of a set of highly optimized implementations for different input matrices. The unrolled matrix dimensions produced by im2col may end up with a sub-optimal matrix multiplication in cuBLAS.

[12] is the first study to observe that lowering convolution to matrix multiplication has high performance in CNNs. Yanqing et al. [39] independently found that im2col-based convolution is efficient in Caffe deep learning framework. By transforming a convolution into a general matrix multiplication, it takes the advantage of highly-tuned linear algebra libraries, e.g., cuBLAS. Caffe’s default implementation calls matrix multiplication iteratively for each image in the mini-batch. In contrast, Gu et al [28] have showed it gains a performance boost of around 4-5 times by using batched im2col over multiple images. The batched scheme increases data parallelism and moves the unrolled matrix size to a more favorable region in BLAS. [29] has also demonstrated that batching up multiple input images against the same kernel matrix once can improve the performance. Indeed, im2col-based approach is BLAS-friendly, the main disadvantage is that it causes significant memory overhead during the im2col process. It involves redundant tensor duplication in the 2d unrolled input tensor. However, as the kernel stride increases, the tensor duplication decreases until none remains when the stride is greater or equal to filter height/width. To address the memory issue, [14] materializes the unrolled matrix lazily to perform the matrix multiplication, which computes on tiles of input and filter tensors. Cho et al. [15] proposed a simple yet novel way to reduce the amount of duplication, while still exploiting the matrix multiplication in BLAS, at the expense of additional matrix multiplication calls. Compared with the conventional im2col-based convolution, both algorithms do not fully materialize the unrolled matrix.

As we compare the algorithm complexity of both direct convolution and im2col based method, they share the same complexity. Since the im2col routine transforms the convolution into a more generic matrix multiplication, which can be easily parallelized by highly efficient cuBLAS library [57]. The direct convolution, on the other hand, can provide a very fast convolutional implementation for some specific convolutional

configurations as we have shown in section 3.1, because it provides specialized optimized implementations to these convolutions, but it suffers from poor performance for other configurations. Compared with the direct convolution used in cuda-convnet [41], [38] observed a speed up of 1.3 on K20.

3.3 FFT-based Convolution

FFT-based convolution makes use of Fast Fourier Transform (FFT) to compute pointwise products in frequency domain, which are equivalent to spatial convolutions based on the convolution theorem.

$$f * g = \mathcal{F}^{-1}(\overline{\mathcal{F}(f)} \cdot \mathcal{F}(g)) \quad (3.1)$$

Where \mathcal{F} and \mathcal{F}^{-1} denote the Fourier Transform and inverse Fourier Transform, and $*$, \cdot , and $\overline{}$ are convolution, complex pointwise product, and complex conjugation, respectively. Although it is well known that convolutions can be computed as pointwise product in the frequency domain, it was not used in the 90’s, possibly because the number of feature maps is small. As in modern CNNs, the number of feature maps is large to amortize the overhead of FFT-based convolution. In this approach, the sizes of both input tensor and weight tensor have to be equal by padding with zeros prior the transformation. Then they are transformed from the spatial domain to the frequency domain with FFT. Following the FFT transformation, we perform a pointwise multiplication between the resulting input FFT transform and the complex conjugate of the filter FFT result. Applying the convolution theorem 3.1 to equation 2.5, it becomes:

$$y_{(n,j)} = \sum_{i \in C} \mathcal{F}^{-1} \left(\mathcal{F}(x_{(n,i)}) \circ \overline{\mathcal{F}(f_{(j,i)})} \right) \quad (3.2)$$

For convolution involving multiple-channel images with multiple kernels described in the above equation, pointwise multiplication can be converted to a complex general matrix multiplication (Cgemm) by transposing the tensors from NCHW to HWNC layout [74], which enable us to utilize the highly-tuned cuBLAS routine in the frequency domain. We then apply an inverse FFT to the result of multiplication to recover the

Table 3.1: Algorithm complexity comparison between direct convolution and FFT-based convolution (See table 2.1 for notations)

	Direct Convolution	FFT-based Convolution
Forward Pass	$N \cdot K \cdot C \cdot R \cdot Q \cdot U \cdot V$	$cHW \log(HW) [K \cdot N + C \cdot N + K \cdot C] + 4N \cdot K \cdot C \cdot H \cdot W$
Back Propagation	$N \cdot K \cdot C \cdot H \cdot W \cdot U \cdot V$	$cRQ \log(RQ) [K \cdot N + C \cdot N + K \cdot C] + 4N \cdot K \cdot C \cdot R \cdot Q$
Gradient Accumulation	$N \cdot K \cdot C \cdot U \cdot V \cdot R \cdot Q$	$cHW \log(HW) [K \cdot N + C \cdot N + K \cdot C] + 4N \cdot K \cdot C \cdot H \cdot W$

convolution in the spatial domain. The output needs to be cropped in order to yield an output with the same size as the direct convolution.

FFT-based approach greatly reduces the algorithmic complexity of convolution in the spatial domain. However, one major drawback is the need for large temporary memory. It has twofold, first, the weight tensor needs to be padded to the same size as input tensor. The memory overhead is high if input tensor is much larger than weight tensor. When the weight size is significantly smaller than the input, too much padding to the input could occur and FFT-based convolution is less efficient. As a consequence, a tiling strategy [74] [34] is used to decompose a large convolution into smaller ones, which can be used to reduce the memory overhead. Second, additional memory is required to store the FFT coefficients. Due to the symmetry property of real inputs in the Fourier space, roughly half of FFT coefficients need to be stored. It is also exploited to reduce the pointwise product computation cost.

Table 3.1 is the algorithm complexity comparison between FFT-based convolution and direct convolution. It requires $U \cdot V \cdot R \cdot Q$ operations to convolve a $H \times W$ image with a kernel of size $U \times V$ by direct convolution because each element in the output image of size $R \times Q$ needs $U \cdot V$ operations. Thus, it takes $N \cdot K \cdot C \cdot R \cdot Q \cdot U \cdot V$ for direct convolution in the forward pass. When compute the convolution by FFT, the FFT time complexity of an image with size $H \times W$ is $cHW \log(HW)$, where c is a constant. It requires $cHW \log(HW) (C \cdot N + K \cdot C)$ operations to transform input

feature maps and kernels to the Fourier domain, and $cHW \log(HW)(K \cdot N)$ operations to transform the output feature maps to the spatial domain. In the Fourier domain, each complex number takes 4 operations, and the total number of operations for pointwise multiplications is $4N \cdot K \cdot C \cdot H \cdot W$. As for back propagation and gradient accumulation, the same analysis can be applied. In the forward pass, direct convolution scales as $\mathcal{O}(N \cdot K \cdot C \cdot R \cdot Q \cdot U \cdot V)$, whereas FFT-based convolution scales as $\mathcal{O}(N \cdot K \cdot C \cdot H \cdot W)$. For large kernel, the FFT-based convolution is obviously more efficient than direct convolution since UV causes the time complexity of direct convolution increases rapidly. Thus, convolution with large kernel size is more efficient with FFT-based convolution.

For large kernel sizes, the FFT-based algorithm is superior compared to other convolution approaches because its complexity does not depend on the kernel size. Although the pointwise product in the frequency domain is much less expensive than convolution in the spatial domain, the input and weight need to be transformed to the frequency domain and the product result needs to be transformed back. For a given convolution, the overhead of FFT-based convolution can outweigh its algorithmic advantage and it may be less efficient, e.g., `fbfft` [74] is less efficient than `cuDNN` [14] if the kernel size is less than 7 from the empirical results in [51]. However, because we compute the convolution between each input image in a mini-batch with each filter, and we only compute the FFT of each input image and each filter once and reuse these FFTs many times to amortize the FFT transform cost, thus the overhead of computing FFTs is greatly reduced. The FFT savings also exist in inverse FFT transform. As in the equation 3.2, we take the summation of IFFT for each elementwise product result to compute the output feature map. Instead, we take the IFFT of this sum once since FFT is a linear operation, which results in considerable savings of IFFT time. Finally, the Fourier coefficients of the output gradients can be reused when backpropagating gradients to the input and filter.

FFT-based convolution is highly dependent on the FFT operation, which can be either based on the off-the-shelf NVIDIA’s `cuFFT` library, or based on specific

implementation of FFT, e.g., `fbfft` [74]. Motivated by the limitation of `cuFFT` library as a black box, and the 2-D convolution is often small and falls out of the regime for which `cuFFT` has been optimized, `ftfft` [74] implemented their own FFT routine and it provides efficient performance over `cuFFT` at sizes 8-64. For the best performance in `cuFFT`, it is worth noting that `cuFFT` performance is sensitive to FFT size and very often slight changes in size result in large performance differences due to different implementations used in `cuFFT`. `cuFFT` programming guide[56] recommends using the size of power of two to achieve the best performance.

3.4 Winograd Convolution

As very small (3×3) convolution filter provides a significant improvement over previous CNN configurations [68], Lavin et al. [44] introduced a fast convolutional algorithm to reduce the complexity using Winograd’s minimal filtering algorithm [77]. It is originally used to compute the output for FIR filters. The causal FIR filters has the following definition:

$$y[n] = \sum_{k=0}^M b_k x[n - k] \quad (3.3)$$

where $x[n]$ and $y[n]$ refer to input and output signal, and M and b_k are the filter order and filter coefficients. Because FIR filter computation is mathematically similar to convolution, Winograd algorithm can be applied to convolution. Its key idea is to reduce the number of expensive multiplication operations by increasing the inexpensive addition operations. To compute m outputs with an r -tap FIR filter, denoted by $F(m, r)$, it requires $\mu(F(m, r)) = m + r - 1$ multiplications [77]. For example, to compute the two outputs for a 3-tap FIR filter with coefficient g_i with a set of elements d_i , $F(2, 3)$, the standard algorithm requires 6 multiplications.

$$\begin{aligned} F_0 &= g_0 d_0 + g_1 d_1 + g_2 d_2 \\ F_1 &= g_0 d_1 + g_1 d_2 + g_2 d_3 \end{aligned} \quad (3.4)$$

In contrast, Winograd algorithm computes these two outputs using 4 multiplication.

$$\begin{aligned}
m_1 &= (d_0 - d_2) g_0 \\
m_2 &= (d_1 + d_2) \frac{g_0 + g_1 + g_2}{2} \\
m_3 &= (d_1 - d_3) g_2 \\
m_4 &= (d_2 - d_1) \frac{g_0 - g_1 + g_2}{2} \\
F_0 &= m_1 + m_2 + m_3 \\
F_1 &= m_2 - m_3 - m_4
\end{aligned} \tag{3.5}$$

While the number of multiplications is reduced from 6 to 4, it is achieved at the expense of additional additions. Compared with the standard algorithm using 4 additions, this method uses 4 additions, 3 additions, and 4 additions for the input data, filter and output data, respectively. Additionally, it uses 2 other multiplications by a constant (multiply 1/2). Nevertheless, considering a multiplication is more expensive in terms of execution time, the benefit brought by multiplication savings is more than the overhead of additional additions and constant multiplication, and Winograd algorithm has potential to yield performance improvements.

The equation 3.5 for $F(2, 3)$ can be written in matrix form as:

$$Y = A^T [(Gg) \odot (B^T d)] \tag{3.6}$$

where \odot represents element-wise multiplication, and the matrices A, B, and G are given as follows:

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \quad G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} \quad A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} \tag{3.7}$$

These matrices can be derived from [77]. Thus, different choices of m and r in $F(m, r)$ need to obtain different matrices A, B and G. As equation 3.6 shows, Winograd algorithm consists of three steps. First, both input and filter are transformed by their

respective matrices G and B^T , and then element-wise multiplication is done. The last step is inverse transformation by matrix A^T .

In the context of convnets, we operate on 2D convolutions. Winograd algorithm can be generalized to 2D filters by nesting one Winograd algorithm $F(m, r)$ with another one. Thus, a 2D convolution $F(m \times n, r \times s)$ to compute $m \times n$ outputs with a $r \times s$ kernel under Winograd algorithm, it requires the following multiplications.

$$\begin{aligned} \mu(F(m \times n, r \times s)) &= \mu(F(m, r))\mu(F(n, s)) \\ &= (m + r - 1)(n + s - 1) \end{aligned} \tag{3.8}$$

As shown in equation 3.8, Winograd convolution of a 3×3 over a 4×4 image resulting in a 2×2 feature map involves $(2 + 3 - 1) \times (2 + 3 - 1) = 16$ multiplications, whereas direct convolution requires $3 \times 3 \times 2 \times 2 = 36$ multiplications. It uses 20 fewer multiplication operations than the direct convolution.

Following the same principles as equation 3.6, the matrix form of 2D Winograd convolution $F(m \times n, r \times r)$ whose input size is $(m + r - 1) \times (m + r - 1)$ and output size is $m \times m$ is as follows:

$$Y = A^T [[GgG^T] \odot [B^T dB]] A \tag{3.9}$$

To implement convolutions in CNNs, each feature map is divided into tiles of size $(m + r - 1) \times (m + r - 1)$, and the neighboring tiles have $r - 1$ overlapped elements. $F(m \times n, r \times r)$ is then used to computed for each tile and kernel combination and results are summed over all feature maps [44].

Although Winograd algorithm computes convolution over small tiles of input data in lower arithmetic complexity, the disadvantage of this approach has twofold. First, the number of operations grows quadratically with kernel size, thus this approach is often used for small kernels (e.g., 3×3) and it has high performance in cases of small kernels and mini-batches [44]. Second, since the magnitude of elements in the transform matrix increases with filter size and it may cause severe roundoff problems in floating point arithmetics [77], and the numerical accuracy of Winograd convolution decreases

as larger kernels are used. Thus, Winograd algorithm can only be used for small kernels. In order to mitigate the numerical inaccuracy, Vincent et al. [75] proposed a simple non-fused approach to implement a 5×5 tile winograd convolution, which means all transform matrices are implemented in separate CUDA kernels. Compared with the prevalent 3×3 tile, this large tile not only provides speedups, but also increases accuracy.

3.5 Strassen-based Convolution

Cong et al. [17] employ the Strassen matrix multiplication [69] to reduce the arithmetic complexity of the redefined matrix multiplication. The Strassen algorithm was originally used to reduce computational workload in matrix multiplication. Let X , Y , and W be square matrices with size of $2^n \times 2^n$. For the following matrix multiplication, we first partition X , Y , and W into equally sized block matrices.

$$Y = W \times X$$

Then we have:

$$Y = \begin{pmatrix} Y_{1,1} & Y_{1,2} \\ Y_{2,1} & Y_{2,2} \end{pmatrix}, W = \begin{pmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{pmatrix}, X = \begin{pmatrix} X_{1,1} & X_{1,2} \\ X_{2,1} & X_{2,2} \end{pmatrix}$$

The normal blocked matrix multiplication would be:

$$\begin{aligned} Y_{1,1} &= W_{1,1} \times X_{1,1} + W_{1,2} \times X_{2,1} \\ Y_{1,2} &= W_{1,1} \times X_{1,2} + W_{1,2} \times X_{2,2} \\ Y_{2,1} &= W_{2,1} \times X_{1,1} + W_{2,2} \times X_{2,1} \\ Y_{2,2} &= W_{2,1} \times X_{1,2} + W_{2,2} \times X_{2,2} \end{aligned}$$

We need 8 multiplications in this representation. Strassen algorithm instead defines new matrices as follows.

$$\begin{aligned}
M_1 &:= (W_{1,1} + W_{2,2}) \times (X_{1,1} + X_{2,2}) \\
M_2 &:= (W_{2,1} + W_{2,2}) \times X_{1,1} \\
M_3 &:= W_{1,1} \times (X_{1,2} - X_{2,2}) \\
M_4 &:= W_{2,2} \times (X_{2,1} - X_{1,1}) \\
M_5 &:= (W_{1,1} + W_{1,2}) \times X_{2,2} \\
M_6 &:= (W_{2,1} - W_{1,1}) \times (X_{1,1} + X_{1,2}) \\
M_7 &:= (W_{1,2} - W_{2,2}) \times (X_{2,1} + X_{2,2})
\end{aligned}$$

$Y_{i,j}$ can be expressed in terms of new matrices M_k .

$$\begin{aligned}
Y_{1,1} &= M_1 + M_4 - M_5 + M_7 \\
Y_{1,2} &= M_3 + M_5 \\
Y_{2,1} &= M_2 + M_4 \\
Y_{2,2} &= M_1 - M_2 + M_3 + M_6
\end{aligned}$$

Here the number of multiplications is reduced from 8 to 7. The algorithm recursively partitions the submatrices into numbers in X , Y and W . Although each recursion reduces the number of multiplications, it incurs additional additions. The number of additions increases from 4 to 18. The overhead of additions may negate the multiplications savings. Cong et al. [17] extend the Strassen algorithm into convolutions in CNNs. It redefines $Y = W \times X$ with the following form.

$$Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_R \end{pmatrix}, W = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1Q} \\ w_{21} & w_{22} & \cdots & w_{2Q} \\ \vdots & \vdots & \ddots & \vdots \\ w_{R1} & w_{R2} & \cdots & w_{RQ} \end{pmatrix}, X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_Q \end{pmatrix}$$

where y_r , w_{rq} and x_q ($r \in [1, R]$, $q \in [1, Q]$) represent output feature maps, convolution kernels, and input feature maps, respectively. Each w_{rq} and x_q pair perform convolutions. This redefined matrix multiplication recursively applies the Strassen algorithm

until the submatrices partitioned into the w_{rq} and x_q pair. The performance gain from this approach root from the large granularities of the w_{rq} and x_q pair. The redefined matrix multiplication provides another view of convolutions in CNNs, and it can reduce the number of operations by up to 47%.

3.6 Summary

In this chapter, we reviewed the major types of convolution approaches in CNNs. These approaches optimize convolution from different algorithmic aspects, and they have clear drawbacks. Both Im2col+GEMM and FFT-based convolutions require a large amount of extra memory. For winograd convolution, numerical accuracy decreases as kernels increase. The overhead of additions may negate the multiplications savings for small granularity input feature map and kernel pairs in strassen-based convolution. In terms of performance in the parameter space, FFT-based convolution is well-optimized for large kernel convolution, whereas winograd convolution is efficient when the kernel size is small.

Matrix multiplication is used by all convolution approaches, with the exception of the direct convolution. Therefore, it is important to provide an efficient matrix multiplication on GPUs. It is recommended to use highly-optimized libraries such as cuBLAS on Nvidia GPUs. Matrix multiplication routine in cuBLAS internally chooses from a set of implementations depending on matrix dimensions. If the matrix dimensions are out of the optimal regime for cuBLAS, the performance of matrix multiplication can not reach the best achievable performance. An alternative approach is to implement specific matrix multiplication, e.g., maxDNN [43] makes use of hardware-specific optimizations on Maxwell Nvidia GPUs, and implements a low-level assembly matrix multiplication. Our proposed convolution also uses a variant of matrix multiplication, which is not immediately available in cuBLAS. Instead, we implement our own matrix multiplication routine. This routine has the potential to be optimized to better squeeze the GPU computational resources; we leave this for future work.

Chapter 4

FINE-GRAINED FFT CONVOLUTION

4.1 Motivation

In this section we explain how data redundancy is discovered from the implementation details of the im2col-based convolution, and how this novel observation motivates the proposed more efficient implementation of convolution.

Recall that the im2col operation reshapes the input feature map as a concatenation of columns stretched by the local patches of the input feature map. The kernels are already stored as a kernel matrix. Therefore, the convolution is transformed to a matrix multiplication to take advantage of highly optimized GEMM libraries. During the im2col process, since the receptive fields overlap, the elements in the overlapped area are duplicated into multiple distinct columns. The duplication of the overlapped elements incurs lots of redundancy. The exact degree of redundancy is hard to predict analytically, however can be measured once all parameters are known. In addition to the redundancy of overlapped elements, there is another kind of redundancy due to the treatment of the input feature map. Sometimes the convolution layer would pad the input feature map with zero to keep the spatial size constant as well as preserve the information at the border. Thus the im2col process also duplicates zeros. We could skip

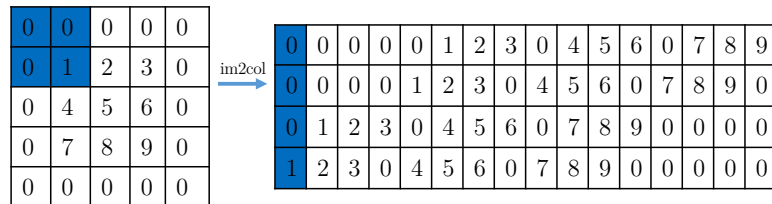


Figure 4.1: A 3×3 input with zero-padding of size 1 convolves with a 2×2 kernel (shown in blue on the left). On the right is a larger matrix generated by the im2col process with redundancy.

multiplications that always give zero in matrix multiply if we know the distribution of zero element after the `im2col` operation. For example, Figure 4.1 shows a 3×3 input with zero-padding size of 1 with respect to a 2×2 kernel are processed to generate an unrolled output by the `im2col` operation.

To exploit the redundancies in `im2col` and develop more efficient convolution algorithm, we need to answer three questions. (1) How to mathematically describe the duplication of the input feature map in `im2col`? (2) How to mathematically describe the distribution of zero elements introduced in zero-padding? And (3) Where are the absolutely necessary elements in the resultant matrix after the `im2col` step? In the remainder of this work, we develop a recursive data pattern to describe the redundancy we find. With the redundancy pattern description, we are able to transform the convolution algorithm to avoid computation and storage on those redundant elements. The final product is our fine-grained FFT convolution algorithm.

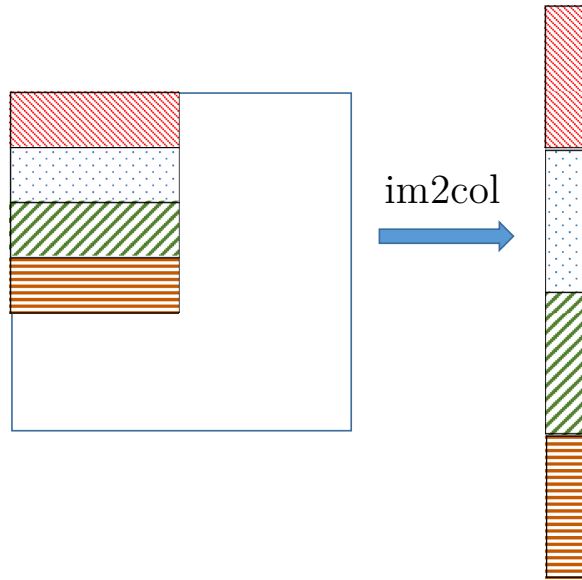


Figure 4.2: The close look of im2col process by reshaping the input patch overlapped with a kernel of four rows into a column.

4.2 A New Data Pattern

In this section, we first take a close look at how im2col works for each row of the kernel, and then show why the particular visiting process introduces two types of redundancy when im2col's kernels slide on CNNs' feature map both horizontally and vertically. We then develop a concise mathematical presentation to describe the redundancy mechanism. Particularly we reveal the connection between the data pattern and the doubly block Hankel matrix, and present a new way to express the data pattern unique to the im2col process. With all these, we present the theoretical foundation of how the convolution in CNNs can be asymptotically optimized.

4.2.1 im2col Process

Let us first take a closer look at the im2col operation since the new data pattern is dependent on the im2col operation. As shown in Figure 4.2, the kernel has four rows indicated by different colors, and the im2col operation transposes and concatenates each row of the overlapping patch in the feature map into one long column. As the kernel slides horizontally and vertically, each row in the kernel works independently.

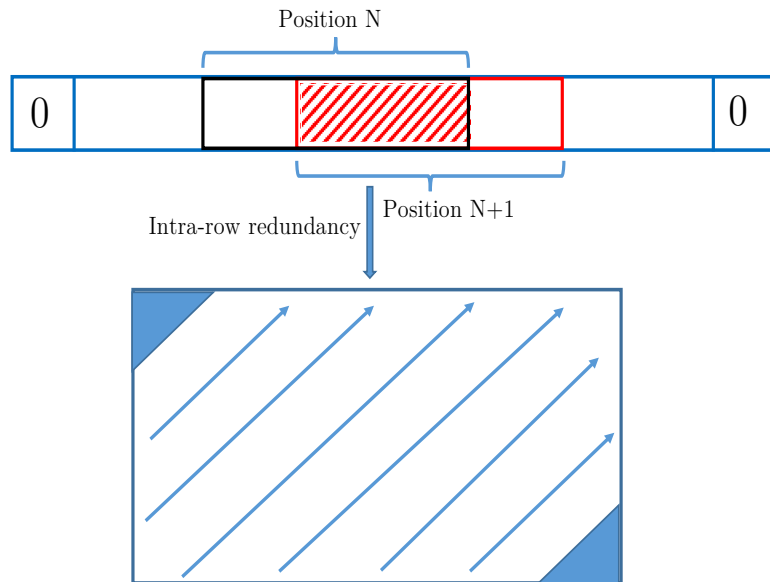


Figure 4.3: The row kernel slides on the one-row feature map, and it incurs intra-row redundancy. The skew diagonals denoted by the blue arrows are constant, and blue triangles represent zero elements.

Alternately, we can think of this 2D filter as a set of 1D row kernel, and the final resultant matrix generated by `im2col` is the composition of results of each row kernel. As the kernel slides horizontally, it incurs the redundancy within the row kernel, which we name it intra-row redundancy.

4.2.2 Intra-row Redundancy

We take one row from both the feature map and filter as an example shown in figure 4.3, and the one-row feature map is padded one zero on both sides. We assume the length of row kernel is n . As the row filter slides along the one-row feature map, the current position $N + 1$ and the previous position N (denoted by red, black rectangles respectively) are overlapped by $n - 1$ (indicated by red striped pattern) and only the leftmost element in the previous position and the rightmost element in the current one are distinct. In the beginning, the row kernel starts at the leftmost position of the feature map. As the filter slides, `im2col` operation transposes the elements in these positions to columns and concatenate them in a shoulder by shoulder manner. Once the row filter finishes sliding to the rightmost, elements in each neighboring of columns

generated by `im2col` are overlapped by $n-1$ and shifted up by one. As can be seen from figure 4.3, the skew diagonals of the resultant matrix denoted by the blue arrows are constant. Because zeros are padded symmetrically on the border, the upper left and the lower right blue triangles are zeros, where their length of sides along height and width directions is equal to p . Let us formally define the intra-row redundancy. Suppose the length of the one-row feature map and row kernel are m and n . Recall that we only consider stride is 1 throughout this dissertation. According to the relation in equation 2.4, the width of resultant matrix is $m - n + 2p + 1$ (p is padding size), and its dimensions are $n \times (m - n + 2p + 1)$. Thus, the intra-redundancy data pattern is as follows:

$$Output_{intra}[i][j] = \begin{cases} Output_{intra}[i-1][j+1], & \text{if } i > 1, j > 0 \text{ and} \\ & i \leq n, j < m - n + 2p - 1 \\ 0, & \text{if } i > 0, 0 < j \leq p - i + 1 \text{ or} \\ & n - p < i \leq n, m - i + 1 < j \leq m \end{cases} \quad (4.1)$$

Where $Output_{intra}$ denotes the output matrix with intra-row redundancy generated by the `im2col` operation when the kernel slides horizontally. Based on the analysis, we observe a data pattern with redundancy as the row filter slides along the feature map and determine how zeros by padding are distributed in the output matrix.

4.2.3 Inter-row Redundancy

We have shown how the sliding of filter horizontally leads to intra-row redundancy, and we focus on analysis of kernel sliding in the vertical direction in this part. In the convolution process, kernel begins from the top left corner and moves a stride size step to the right. In this process, rows in the kernel independently incurs the redundancy pattern as figure 4.3. When the kernel reaches the right border of feature map, it shifts one row vertically and continues to slide from left to right, which the other redundancy incurs because each row kernel will traverse the elements the next row kernel just traversed. Hence, we name it inter-row duplication. The kernel continues to go through the same process until it finishes at the bottom right corner of the feature map. In figure 4.4, the kernel has four rows indicated by different colors and

the feature map is padded zeros with size one. As the kernel reaches the bottom right corner of the feature map, k_1 , k_2 , k_3 and k_4 independently generate row from one to four on the output matrix and each block has intra-row redundancy following the same principle in section 4.2.2. Since k_1 will traverse the elements in the feature map k_2 just traversed as k_1 shifts one row vertically, and this is where inter-redundancy incurs. The inter-row redundancy also follows the similar pattern as the intra-row redundancy that blocks on skew diagonals are the same, which is indicated by blue dotted arrows in figure 4.4. Since we only pad zeros with size one and the row kernel slides the padded rows that are entirely zeros, the upper left and lower right blocks are zero matrices. We set the feature map and kernel sizes to be $m \times m$ and $n \times n$, padding the feature map with zeros of size p around the borders. The `im2col` operation rearranges the feature map into dimensions of $n^2 \times (m - n + 2p + 1)^2$, and each block in the output matrix is $n \times (m - n + 2p - 1)$.

$$Output_{inter}[i][j] = \begin{cases} Output_{inter}[i-1][j+1], & \text{if } i > 1, j > 0 \text{ and} \\ & i \leq n, j < m - n + 2p - 1 \\ 0, & \text{if } i > 0, 0 < j \leq p - i + 1 \text{ or} \\ & n - p < i \leq n, m - i + 1 < j \leq m \end{cases} \quad (4.2)$$

where i and j are the indices for the blocks in the output matrix $Output_{inter}$. Within each block that is not zeros, it follows the data pattern shown in equation 4.1. If $n = m - n + 2p + 1$, the output matrix generated by `im2col` is a doubly block Hankel matrix, which we will describe the details in subsection 4.2.6. It is worth noting that if the padding size is zero the distribution of zeros in the new feature map does not exist, however, if the padding size is large, and the output matrix has more zero blocks allowing for potential optimization that skips the multiplication with zero blocks.

4.2.4 Im2col-based Convolution Redundancy

In CNNs, convolutional layers perform batched convolution that each input feature map convolves spatially with a kernel and the results of the convolutions are summed across feature maps to produce an output feature map. In this case, `im2col`

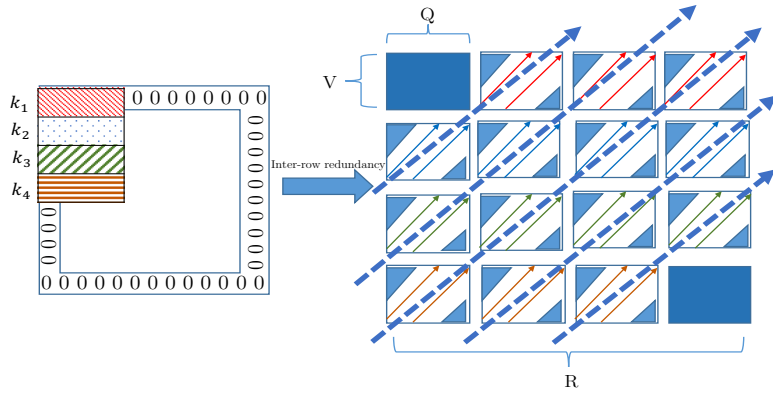


Figure 4.4: As the kernel traverses the entire feature map, the output matrix generated by `im2col` has inter-row redundancy denoted by dotted blue arrows. Each block is $V \times Q$ with total number of $K \times R$. Refer to table 2.1 for notations.

duplicates each feature map in the same manner and concatenates small output matrices into a larger output matrix, and each small output matrix is depicted in figure 4.4. The intra-row and inter-row redundancy follow the same data pattern that elements or blocks on the skew diagonals are the same. Recall that in section 3.2, the input data with dimension NCHW is unrolled to N matrices with dimensions $CUV \times NRQ$ by a kernel matrix with dimensions $K \times CUV$. The larger output matrix has C small output matrix, and each small $UV \times NRQ$ matrix has the data pattern described in equation 4.2.

In this part, we use a bottom-up approach to systematically study the redundancy in the `im2col` process. We analyze how `im2col` incurs redundancy from 1D and 2D convolutions to batched 2D convolution in CNNs as well as the zero distribution by padding. The zero distribution provides us an opportunity to skip the multiplication and save the computation resources, and the redundancy motivates us to design new algorithms to avoid it to save the memory consumption, but more importantly provides us with an optimization opportunity for convolution in CNNs.

4.2.5 Padded Zero Distribution

A convolutional layer's output size depends on the input size, kernel size, striding as well as zero-padding. It allows users to surround the input with zeros, which can

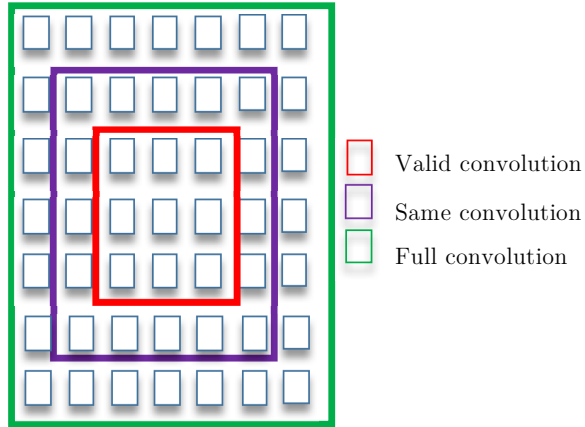


Figure 4.5: Three 2D convolution types: valid, same and full convolutions. Each of them is denoted by red, purple and green colors respectively. The input size is 5×5 and the kernel size is 3×3 .

control the output size and preserve the information at the borders. Without padding, the input size is shrunk after each convolutional layer and the information at the borders is lost. Based on the zero-padding size, 2D convolution is categorized into three types: valid convolution, same convolution and full convolution. Valid convolution does not use any padding on the input, and the same convolution zero-pads the input such that the output size is the same as the input after convolution. Both of them does not increase the output size with respect to the input size. In contrast, the full convolution increases the output size with proper zero-padding. Recall that the output width is defined as $Q = (W - V + 2P)/S + 1$ in equation 2.4, and stride size is 1. For the full convolution, the padding size P is $V - 1$, while P is $(V - 1)/2$ for the same convolution. It is also called half convolution because the padding size is nearly half of the kernel size. Figure 4.5 shows an example of a 3×3 kernel convolving over a 5×5 input for these three convolution types. In this example, the input size is 5×5 , and the kernel size is 3×3 . Because no padding is applied on the input for the valid convolution, the output size is 3×3 , which is denoted by the red square. Zero-padding sizes are 1 and 2 for same and full convolutions respectively, denoted by purple and green squares in the figure. As we have discussed in the previous sections, the padded zeros are duplicated in the im2col process. In the inter-row redundancy, the padded zeros are

distributed at the upper left and bottom right corners of the unrolled feature maps. It allows us to skip the FFT computation and block matrix multiplication on the zero blocks. According to equation 4.2, we plug the padding size $(V - 1)/2$ and $V - 1$ into the equation, the padded zero distribution is trivial to compute for same and full convolutions.

4.2.6 Doubly Block Hankel Matrices

With the intra-redundancy and the inter-redundancy revealed, we can now answer the questions asked in section 4.1 as to how to express and use the redundancy patterns.

The `im2col` process converts the input patches to columns and forms an unrolled matrix, which has the inter-row redundancy, whereas each block has intra-row redundancy, as demonstrated in figure 4.6. The padded zero elements are distributed in the upper-left and lower-right blocks of the unrolled matrix. Additionally, they are distributed within the upper-left and lower-right corners of each non-zero blocks. Each row in the input feature map is distributed to the first column and last row within the blocks depicted in figure 4.6, because a new element is shifted up in intra-row redundancy. We now are able to track all the elements from the input feature map and fully reveal the data pattern introduced by the `im2col` process.

We make a key finding here. The data pattern, which features both intra-row and inter-row redundancy, can be qualitatively summarized as following. Each block along skew-diagonals in figure 4.4 are identical, while each block has intra-row redundancy that elements along skew-diagonals are constant. The equations 4.2 4.1 that summarize the intra-row and inter-row redundancy patterns is in fact the definition for a Hankel matrix which each value along skew-diagonals are constant. The output matrix generated by `im2col` for an input feature map is actually a *doubly block Hankel matrix*. Each individual block in the matrix is a Hankel matrix, and the whole matrix with respect to its blocks is also a Hankel one. Hankel matrix is a matrix in which each

values along ascending skew-diagonals are constant. An $n \times n$ Hankel matrix takes the form:

$$H = \begin{pmatrix} h_1 & h_2 & \dots & h_{n-1} & h_n \\ h_2 & h_3 & \dots & h_n & h_{n+1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ h_{n-1} & h_n & \dots & h_{2n-3} & h_{2n-2} \\ h_n & h_{n+1} & \dots & h_{2n-2} & h_{2n-1} \end{pmatrix} \quad (4.3)$$

Its elements are determined by a $2n - 1$ length sequence $\{h_i | 1 \leq i \leq 2n - 1\}$.

$$H_{Block} = \begin{pmatrix} H_{11} & H_{12} & \dots & H_{1N} \\ H_{12} & H_{22} & \dots & H_{2N} \\ \vdots & \vdots & \vdots & \vdots \\ H_{1N} & H_{2N} & \dots & H_{NN} \end{pmatrix} \quad (4.4)$$

In the matrix H_{Block} , all H_{ij} are Hankel matrices. The structure of H_{Block} , with respects to its sub-matrices is also Hankel, thus H_{Block} is a doubly block Hankel matrix.

In CNNs, the kernel matrix has dimensions $K \times CUV$, and the input data with dimension $NCHW$ is unrolled to N matrices with dimensions $CUV \times RQ$. Each matrix has C sub-matrices, and each of them is a doubly block Hankel matrix with size of $UV \times RQ$ (see table 2.1 for notations). Using the doubly block Hankel matrix to represent the feature map matrix generated by the `im2col`, the convolution actually become the multiplication between Hankel matrix and vector. Due to the intrinsic data redundancy in Hankel matrices, such Hankel-matrix-vector multiplication has theoretically lower computational complexity than a normal matrix-vector multiplication. We use the Fast Fourier Transform (FFT) to asymbolotically optimize the Hankel-matrix-vector multiplication and therefore improve the performance of the convolution in CNNs. In the next section, we will present our proposed fine-grained FFT-based convolution in details.

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 4.6: A new data pattern revealed in figure 4.1 that each block with the same color are the same, and element along the skew diagonals are constant. The elements bounded by red lines correspond to individual rows from the input feature map.

4.3 Fine-Grain-FFT-Based Convolution Algorithm

We have shown that im2col-based convolution unrolls each input feature map to a doubly block Hankel matrix, and the convolution is transformed to a matrix multiplication between the kernel matrix and doubly block Hankel matrices. Disregarding such redundancy, existing implementations of the im2col+GEMM convolution approach such as that in cuDNN all directly compute the kernel and unrolled input matrices multiplication using BLAS libraries such as cuBLAS. In this section, we show how the Hankel matrix data pattern enables the use of FFT to more efficiently compute the specific matrix multiplication. We then introduce the complete FFT Hankel matrix vector multiplication algorithm in the context of convolution. At last, we demonstrate analytically that our fine-grain FFT based convolution algorithm not only reduces the computational complexity by FFT, but also reduces the memory overhead because it eliminates the needs to fully unroll the input data and replaces unrolling with the implicit element-wise matrix multiplication.

4.3.1 Fast Fourier Transform

The Fast Fourier Transform (FFT) is one of the most important numerical algorithms, which is an efficient procedure to compute the Discrete Fourier transform (DFT). DFT is used to transform the signal from time or spatial domains to frequency domain. It has a wide range of applications, e.g., image processing. Image convolution in the spatial domain can be transformed into a more computationally efficient multiplication in the frequency domain. For an input data x_n of length N , 1D DFT for this input data is defined as follows.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i k n}{N}} \quad k = 0, 1, \dots, N - 1 \quad (4.5)$$

In equation 4.5, each X_k involves a summation over all input data. Therefore, it requires $O(N^2)$ operations and the complexity grows quadratically with N if we compute directly for all 1D input. In contrast, FFT recursively decomposes into smaller DFTs, and the divide and conquer approach reduces the complexity to $O(N \log N)$. Not only FFT

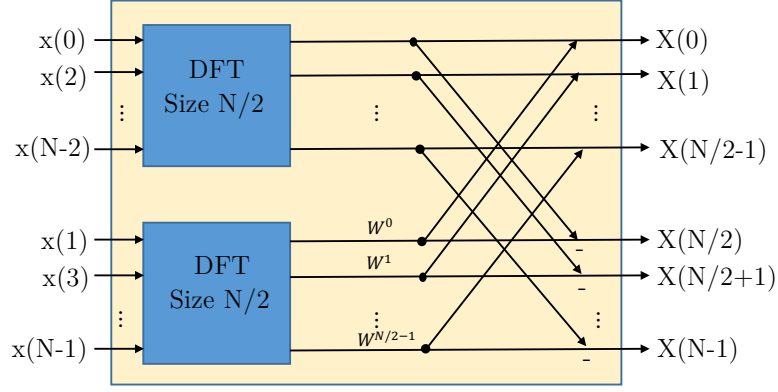


Figure 4.7: Illustration of radix-2 DIT FFT algorithm

is more efficient than DFT, but it is also more accurate than DFT, because fewer operations undertaken results in less round-off error. There are many different FFT algorithms, and one of the most widely used algorithms is the Cooley-Tukey algorithm [18]. It decomposes recursively the DFT of a composite size $N = N_1 N_2$ into N_1 smaller DFTs of size N_2 . As the simplest form of the Cooley-Tukey algorithm, the radix-2 decimation-in-time (DIT) FFT algorithm decomposes N ($N = 2^r$) into two $N/2$ DFTs of even-numbered inputs and odd-numbered inputs. Equation 4.5 becomes:

$$\begin{aligned} X_k &= E_k + e^{-\frac{2\pi i k}{N}} O_k \\ X_{k+\frac{N}{2}} &= E_k - e^{-\frac{2\pi i k}{N}} O_k \end{aligned} \quad (4.6)$$

where $k = 0, 1, \dots, \frac{N}{2} - 1$, E_k and O_k denote the DFTs of even-numbered inputs and odd-numbered inputs respectively. Therefore, the DFT of size N is expressed as two DFTs of size $N/2$, followed by a combination step that corresponding size-2 DFTs are merged into the final DFT result. It is illustrated in figure 4.7. As the decomposition continues recursively, the FFT algorithm decomposes the DFT into $\log N$ stages. Thus the total computational complexity is $O(N \log N)$.

The Fourier transform of a 1-D real-valued input has Hermitian symmetry (conjugate complex symmetry), This property enables us to derive an efficient fine-grained FFT convolution implementation. Specifically, having the input sequence x_i of an even length N , the N input data points produce $N/2+1$ complex output points, and the

remaining $N/2-1$ output points are just the complex conjugates of the corresponding output points.

Recall that the DFT definition is

$$X(k) = DFT[x(n)] = \sum_{n=0}^{N-1} x(n)W_N^{kn}, 0 \leq k \leq N-1$$

where $W_N = e^{-j\frac{2\pi}{N}}$. According to Euler's formula, $W_N = e^{-j\frac{2\pi}{N}} = \cos\left(\frac{2\pi}{N}\right) - j \sin\left(\frac{2\pi}{N}\right)$

When k is 0

$$W_N^{0n} = e^{-j\frac{2\pi}{N}0} = \cos(0) - j \sin(0) = 1$$

$$X(0) = DFT[x(n)] = \sum_{n=0}^{N-1} x(n)W_N^{0n} = \sum_{n=0}^{N-1} x(n)$$

The first output point is the summation of all input data points.

When k is $N/2$

$$W_N^{\frac{N}{2}n} = e^{-j\frac{2\pi}{N}\frac{N}{2}n} = \cos(n\pi) - j \sin(n\pi) = (-1)^n$$

$$X\left(\frac{N}{2}\right) = DFT[x(n)] = \sum_{n=0}^{N-1} x(n)W_N^{\frac{N}{2}n} = \sum_{n=0}^{N-1} x(n)e^{-jn\pi} = \sum_{n=0}^{N-1} x(n)(-1)^n$$

As for other values of k

$$W_N^{kn} = e^{-j\frac{2\pi}{N}kn} = \cos\left(\frac{2\pi}{N}kn\right) - j \sin\left(\frac{2\pi}{N}kn\right)$$

$$W_N^{(N-k)n} = e^{-j\frac{2\pi}{N}(N-k)n} = \cos\left[\frac{2\pi}{N}(N-k)n\right] - j \sin\left[\frac{2\pi}{N}(N-k)n\right]$$

$$= \cos\left(\frac{2\pi}{N}kn\right) + j \sin\left(\frac{2\pi}{N}kn\right) = (W_N^{kn})^*$$

$$X(N-k) = \sum_{n=0}^{N-1} x(n)W_N^{(N-k)n} = \sum_{n=0}^{N-1} x(n) (W_N^{kn})^* = X^*(k)$$

where the star denotes conjugation. Thus for the input with even length, the DFT output is conjugate symmetric. We will exploit this property in subsection 4.3.5.

4.3.2 Hankel Matrix Vector Multiplication

Recall that for $N H \times W C$ -channel input images, `im2col` transforms them to NC doubly block Hankel matrices of size $UV \times RQ$ with Hankel blocks of size $V \times Q$.

The corresponding dimensions of the kernel matrix are $K \times UVC$ for K kernels with spatial dimensions $U \times V$ since it is required that the input tensor and the set of K kernels have the same depth size C in a CNN convolution layer. Here we first show a fast multiplication to multiply a vector v of size $1 \times V$ from the kernel matrix with the Hankel block \mathcal{H} of size $V \times Q$, and then in section 4.3.4 extend it to the doubly block Hankel matrix method.

Hankel matrices are referred as structured matrices, which can be described without loss of information much more concisely than the n^2 elements in $n \times n$ matrices. The immediate benefit is that the storage complexity can be significantly reduced. More importantly for the convolution algorithm, much lower computational complexity for structured matrix-vector product can be obtained via fast matrix-vector products by FFTs. The Hankel block \mathcal{H} can be embedded into a $2Q \times 2Q$ circulant matrix X , and the multiplication by the kernel vector v can be achieved by FFTs, reducing the computational complexity from $\mathcal{O}(Q^2)$ to $\mathcal{O}(2Q \log 2Q)$ operations. To efficiently compute the multiplication, recall the circulant matrix definition in section 2.5, it can be completely specified with only the first row, which is also known as the generating vector x . It is diagonalized by the Discrete Fourier Transform (DFT) matrix regardless of vector x [26]. It can be expressed as

$$X = \mathbb{F} \Delta \mathbb{F}^{-1}$$

where \mathbb{F} is the DFT matrix and Δ is a diagonal matrix containing the eigenvalues of X such that $\Delta = \text{diag}(\mathbb{F}x)$. Therefore, multiplying the circulant matrix X with the kernel vector v is as follows.

$$X\hat{v} = \mathbb{F} \Delta \mathbb{F}^{-1} \hat{v} = \mathbb{F}(\mathbb{F}x \circ \mathbb{F}^{-1} \hat{v}) \tag{4.7}$$

where \circ denotes the Hadamard element-wise vector multiplication and $\hat{v} = (v_V, v_{V-1}, \dots, v_1, 0, \dots, 0)$. It first computes a DFT $\mathbb{F}x$ and an IDFT $\mathbb{F}^{-1} \hat{v}$ and then a final DFT $\mathbb{F}(\mathbb{F}x \circ \mathbb{F}^{-1} \hat{v})$. These three DFTs can be computed efficiently by applying FFTs. Their computational complexity is $O(2Q \log 2Q)$, thus our fine-grained FFT algorithm works at $O(2Q \log 2Q)$ granularity.

4.3.3 Hankel Matrices to Circulant Matrices

The input feature maps are unrolled into doubly Hankel matrices in the `im2col` process and the hankel matrix vector multiplication can be performed using FFT. The first step is to embed hankel matrices into circulant matrices. In this section, we present the details of the procedure on embedding hankel matrices to circulant matrices.

A standard method is to permute the hankel matrix into a toeplitz matrix by multiplying a permutation matrix. The columns in the hankel matrix are permuted left-to-right, and it is converted to a toeplitz matrix. Then the toeplitz matrix is embedded into a larger circulant matrix to achieve fast computation by FFT. The problem of this method is multiplications of the permutation matrix P with all entries are zero except those one on the anti-diagonal are one, which unnecessary operations are performed and GPU computational resources are wasted on the operations. To alleviate this problem, we notice that in fact we only need the generating vector for the circulant matrix, the circulant matrix is not necessarily formed.

$$P = \begin{pmatrix} 0 & 0 & \dots & 0 & 1 \\ 0 & 0 & \dots & 1 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 1 & \dots & 0 & 0 \\ 1 & 0 & \dots & 0 & 0 \end{pmatrix}$$

Hankel matrix 4.3 is permuted into the following toeplitz matrix in which elements on the diagonal are constant, and then embedded into the circulant matrix in 4.9.

$$T = \begin{pmatrix} h_n & h_{n-1} & \dots & h_2 & h_1 \\ h_{n+1} & h_n & \dots & h_3 & h_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ h_{2n-2} & h_{2n-3} & \dots & h_n & h_{n-1} \\ h_{2n-1} & h_{2n-2} & \dots & h_{n+1} & h_n \end{pmatrix} \quad (4.8)$$

$$C = \begin{pmatrix} h_n & h_{n-1} & h_{n-2} & \dots & h_1 & h_{2n-1} & h_{2n-2} & \dots & h_{n+1} \\ h_{n+1} & h_n & h_{n-1} & \dots & h_2 & h_1 & h_{2n-1} & \dots & h_{n+2} \\ h_{n+2} & h_{n+1} & h_n & \dots & h_3 & h_2 & h_1 & \dots & h_{n+3} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \\ h_{2n-1} & h_{2n-2} & h_{2n-3} & \dots & h_n & h_{n-1} & h_{n-2} & \dots & h_1 \\ h_2 & h_1 & h_{2n-1} & \dots & h_{n+2} & h_{n+1} & h_n & \dots & h_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \\ h_{n-1} & h_{n-2} & h_{n-3} & \dots & h_{2n-1} & h_{2n-2} & h_{2n-3} & \dots & h_n \end{pmatrix} \quad (4.9)$$

where the leading $n \times n$ matrix is the toeplitz matrix 4.8. The first row c is the generating vector of the circulant matrix C .

$$c = (h_n \quad h_{n-1} \quad h_{n-2} \quad \dots \quad h_1 \quad h_{2n-1} \quad h_{2n-2} \quad \dots \quad h_{n+1}) \quad (4.10)$$

All the elements in the first row contains two parts. The first part is elements from rows in the original feature maps. The second part is the padded zeros which convert non-square toeplitz matrices to square toeplitz matrices. It simply pads zeros to non-square toeplitz matrices as long as elements along the diagonal are constant. Taking FFTs over the generating vector in equation 4.7, the hankel matrix vector multiplication can be performed using FFTs. To achieve the best performance of cuFFT, it is recommended that the transform size of FFT to be a power of two. Thus the dimension of the square matrix in 4.8 is a power of two, and the size of generating vector of the circulant matrix in 4.10 is also a power of two. We simply need to pad additional zeros to increase the size of toeplitz matrix to the next power of two.

4.3.4 Implicit Element-Wise Matrix Multiplication

The previous section explains the transformation of the unrolled input feature map into a doubly block Hankel matrix data pattern and the use of FFT to optimize the Hankel matrix vector multiplication. In this section we develop an efficient implementation of the Hankel block matrix multiplication. In particular, we present a

unique optimization technique that is derived from the linearity of the DFT. Because DFT is a linear transformation, the linearity allows the sum of the elementwise product directly in the Fourier domain, which leads to considerable savings of FFT time.

We first briefly overview the existing techniques used in the implementation of the im2col-based convolution. Caffe’s default implementation calls matrix multiplication iteratively for each image in the mini-batch. In contrast, Gu et al [28] have showed a performance boost of around 4-5 times is obtained by using batched im2col over multiple images. The batched scheme increases data parallelism and moves the unrolled matrix size to a more favorable region in BLAS. Hadjis et.al. [29] demonstrated that batching up multiple input images against the same kernel matrix once can improve the performance. In this work we adopt the batched scheme that the kernel matrix multiply all the input feature maps in a batch in the Fourier domain.

Comparing to the batching mode, a more important performance issue is the handling of the data redundancy and its derivative memory/computation overhead. All prior work requires unrolling the feature map completely or partially. In our work, we find that the redundant data pattern make it possible to not unroll the input matrix at all for the computing of convolution. It is because a doubly block Hankel matrix can be fully specified by only the distinct elements, e.g., elements bounded by red lines are distinct elements in figure 4.6. The key insight is that the Hankel block matrix has an interesting property that the generating vector for the circulant matrix happens to contain all the distinct elements, and the rest elements are padded. This interesting structure of the circulant matrix makes it possible to compute the matrix vector product by only using the generating vector. Furthermore, these distinct elements are extracted by the im2col process from each individual row of the original feature map, as demonstrated by the red lines in figure 4.6 and the original feature map in figure 4.1. Therefore we don’t need the completely unrolled input matrix, as required in existing approaches, for the computing of convolution. All these reduce not only the memory foot-print but also the computational complexity of the proposed convolution method.

Specifically in implementation, we develop an implicit element-wise matrix multiplication strategy in the Fourier domain. It uses an indexing arithmetic to load the corresponding Fourier coefficients from input feature maps without unrolling them. Accordingly, our approach avoids the FFT computations of redundant Hankel blocks, which in turn reduces the storage of the Fourier coefficients for the redundant Hankel blocks. Compared with a fully unrolled input feature map with size of $H \times W$, it requires $U \times R$ FFT computations of Hankel blocks, whereas our approach only needs $U + R - 1$.

4.3.5 FFT Hermitian Symmetry

We take advantage of another property of FFT to further optimize both the memory storage and the operation complexity of our convolution method. The Fourier transform of a real-valued input is Hermitian symmetric (conjugate complex symmetry). The symmetry allows us to store roughly half Fourier representations to carry out the complete FFT. For even N real-valued input $x_i, i \in \{0, \dots, N-1\}$, the Fourier representations of X_0 and $X_{N/2}$ are real-valued, and X_1 through $X_{N/2-1}$ are the complex conjugates of $X_{N/2+1}$ through X_{N-1} . Thus, we only need to store $N/2 + 1$ complex numbers. Furthermore, we can use the symmetry property to reduce the number of products in element-wise multiplication by almost half. Specifically, the second half of element-wise multiplication can be constructed by simply taking the complex conjugate of the first half. Each element-wise product in Fourier domain requires four multiplications for two complex numbers. Using Gauss' multiplication algorithm [54], the number of multiplications is reduced from four to three. For two complex numbers $a + ib$ and $c + id$, it first computes $t_1 = c * (a + b)$, $t_2 = a * (d - c)$ and $t_3 = b * (c + d)$, and the real and imaginary parts of the result can be computed as $t_1 - t_3$ and $t_1 + t_2$, respectively. Similarly, each element-wise multiplication of complex numbers is replaced by three element-wise multiplication of real numbers.

Table 4.1: Algorithm complexity comparison between FFT-based convolution and our method (See table 2.1 for notations)

	RegularFFT	FineGrainedFFT
Input transform	$2W^2 \cdot \log W \cdot N \cdot C$	$2W^2 \cdot \log 2W \cdot N \cdot C$
Kernel transform	$2W^2 \cdot \log W \cdot K \cdot C$	$2W \cdot \log 2W \cdot K \cdot C \cdot V$
Element-wise multiplication	$W^2 \cdot N \cdot K \cdot C$	$2W^2 \cdot N \cdot K \cdot C \cdot V$
Output inverse transform	$2W^2 \cdot \log W \cdot N \cdot K$	$2W^2 \cdot \log 2W \cdot N \cdot K$

4.3.6 Overall Working Flow

Overall the proposed convolution method is implemented in four steps:

Step 1 Input transform. Since the generating vector of circulant matrix is already contained in each row of input feature maps, we apply 1D FFTs to each row to transform the input. For the best performance in cuFFT, it is worth noting that cuFFT performance is sensitive to FFT size. Very often slight changes in size result in large performance differences due to different implementations used in cuFFT. Therefore we use input padding to find the best cuFFT case for our particular FFT problem instances.

Step 2 Kernel transform. To transform the kernel into Fourier domain, we decompose the kernel matrix into $K \cdot U \cdot C$ tiles, and perform $K \cdot U \cdot C$ 1D FFTs using the batch mode provided by cuFFT.

Step 3 Element-wise computation. For this step, where block matrix multiplication with element-wise product is performed since the doubly Hankel matrix is already partitioned into Hankel blocks. Within each block, we perform element-wise product.

Step 4 Inverse transform. Lastly, inverse FFT transform is performed on the output matrix from Step 3. Only $1 \times Q$ element in the $1 \times 2Q$ output is valid for the $1 \times V$ vector and $V \times Q$ Hankel matrix multiplication, the rest is discarded.

4.3.7 Arithmetic Complexity Analysis

Next we compare the computation complexity of the proposed fine-grained FFT based convolution against the existing regular approach.

Both regular and fine-grained FFT approaches perform convolutions in four basic steps: input transform, kernel transform, element-wise multiplication, and inverse transform. It is worth noting that the element-wise multiplication step in the fine-grained FFT approach is in fact matrix multiplication with element-wise product. As a point of comparison, we could treat it as the element-wise multiplication. The first two steps transform inputs and kernels from a spatial domain to Fourier domain. The third step can be converted to a batched complex general matrix multiplication (Cgemm) for the regular FFT approach. The inverse transform step converts the results back to the spatial domain. Assume we have (N, C, H, W) inputs and (K, C, U, V) kernels, the respective complexity for both approaches for all four steps is listed in table 4.1.

As indicated in this table, the complexity of each step for the regular FFT convolution does not depend on the kernel size. It is expected that the regular FFT approach performs the same regardless of the kernel size since it zero-pads the kernel to be the same size as the input image before applying the FFT. For large kernel sizes, it likely performs better. On the other hand, the complexity of our method depends on the kernel size, which is suited to small kernel convolutions. Table 4.1 shows that the input transform and the final inverse transform have equal number of operations in either of the regular or the proposed methods, respectively. However, our kernel transform needs fewer operations than the regular-FFT approach. With regard to the element-wise multiplication step, our method requires more operations. Thus, the cost of element-wise multiplication may outweigh the algorithmic advantage of fine-grained FFT method for large kernel convolution. Our method would be advantageous when the kernel size is small. The benefits of our method are mainly rooted from its smaller granularity of FFTs $2Q$ since we exploit the redundant data pattern. In contrast, the regular-FFT approach takes FFT2D operation on each feature map, and its FFT granularity is HW .

4.3.8 Autotuning

We apply a simple autotuning strategy to tune our implementation on GPU. Basically our autotuning selects the best configuration of CUDA thread and block parameters for given constraints of input settings and hardware resources. The result of the autotuning can be stored locally and re-used when a similar configuration of inputs passed to the implementation. Since the fine-grained FFT consists of four major steps, input and kernel FFTs, point-wise multiplication, and inverse FFTs. We use off-the-shelf NVIDIA's cuFFT library to carry out FFT. which is a black box library that we have no control over the CUDA parameters. We thus autotune the point-wise multiplication step to find the optimal combination of CUDA parameters, BLOCK_SIZE and NUM_BLOCKS, which represent the thread block size and the number of thread blocks, respectively. The autotuning strategy explore different BLOCK_SIZE and NUM_BLOCKS combinations, where BLOCK_SIZE $\in [32, r]$ and it is a multiple of warp size (32), where r is $\text{NextPowerTwo}(2Q)/2+1$ and NextPowerTwo is a function to find the next power of two. The maximum number of threads per block is 1024 for GPUs with compute capability 6.1, and r is less than 1024 and within the allowed range. NUM_BLOCKS's range is defined as $[1, N \times K \times R]$. For each BLOCK_SIZE and NUM_BLOCKS combination, we use loops to iterate over the matrix data. For threads within each thread block, the upper limit of the loops is r, and the stride of the loop is BLOCK_SIZE, so the threads loop over all of r elements. Similarly, thread blocks loop over the entire matrix data with size $r \times N \times K \times R$, which is the maximum number of threads in the grid. In order to prune the search space to a manageable size, we impose a maximum of 100 steps in the NUM_BLOCKS range as a constraint. Thus we do not exhaustively execute and measure each combination in the search space and the time spent on autotuning is reduced.

4.3.9 Memory Consumption Analysis

The fine-grained FFT method does not fully unroll the input feature maps as the im2col+MM does, which can save memory consumption. In this part, we compare

the memory consumption of our method against the im2col+GEMM method.

The input feature maps are fully unrolled in the im2col+GEMM method. As the kernel convolves over the feature maps, the local patches of feature maps are unrolled into columns, and the memory requirement for unrolled feature maps grows quadratically with kernel size. On the other hand, the fine-grained FFT method does not fully unroll the input feature maps. Each row in the feature maps is padded to $2Q$. To be more specific, the padded size is $\text{NextPowerTwo}(2Q)$. For a $NCHW$ input, the fine-grained FFT method requires $N \times C \times H \times 2Q$ memory footprint. In contrast, the im2col+GEMM method requires $N \times C \times U \times V \times R \times Q$. Furthermore, as the Fourier transform of a real-valued input has Hermitian symmetry, we only store about half the complex entries. A complex entry is twice the size of a float type. Thus the fully unrolled method has $UVR/2H$ times more memory consumption than our method for a NCHW input.

4.4 Evaluation and Performance Analysis

Parameters	Values
CUDA capability major/minor version number	6.1
Total amount of global memory	12GB
Warp size	32
CUDA cores	3840
Total number of registers available per block	64KB
Total number of shared memory available per block	48KB

Table 4.2: Properties of the NVIDIA GeForce GTX TITAN Xp.

We evaluate the proposed convolution method from four aspects: (1) accuracy of result, (2) kernel-level performance comparison with NVIDIA’s cuDNN library [14], (3) application-scenario performance with networks developed in Caffe[39], a leading deep learning programming framework, by replacing Caffe’s own convolution method, and (4) performance profiling to analyze and understand the source of performance improvement. We measure the relative performance of the regular FFT convolution and our proposed approach in terms of speedup. It is defined as the ratio of the execution time of the regular FFT approach to the time taken by the fine-grained FFT one. The larger speedup it achieves, the greater performance improvements our approach has.

Each performance point is the average of five runs. Since the performance of convolution is independent to input values, we randomly generate inputs and use the same input for each data point. The versions of cuDNN and Caffe are 7.1 and 1.0, respectively. Hardware-wise, all the experiments are performed on Nvidia Titan Xp GPU. Detailed parameters of the Titan Xp are shown in Table 4.2. In our experiments, CPU only serves as the command processor and has a negligible impact on performance.

4.4.1 Accuracy

We first measure the accuracy of our method by comparing the results of our convolution with those computed by the im2col+GEMM approach. Table 4.3 shows

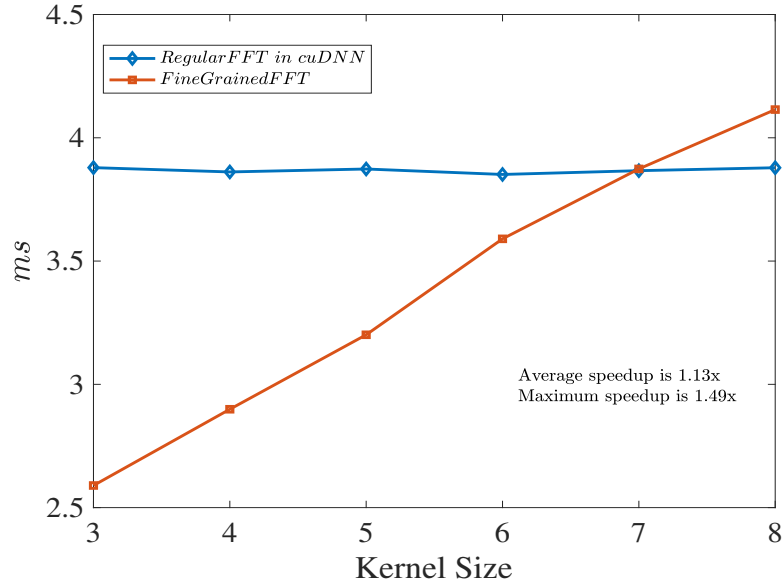


Figure 4.8: RegularFFT and FineGrainedFFT performance comparison as the kernel size varies from 3 to 8 on synthetic benchmark

the numeric accuracy of fine-grained FFT convolution using the convolution configurations listed in the top row. The error is in the order of 10^{-11} , which means that our optimization technique maintains the numerical integrity of convolution.

(U, K, S, P)	$(3, 10, 1, 2)$	$(5, 32, 1, 2)$	$(4, 10, 1, 2)$
Error	4.73E - 11	3.00E - 11	2.38E - 11

Table 4.3: FineGrainedFFT convolution absolute element error. Ground truth is computed by im2col+GEMM convolution.

4.4.2 Kernel-level Performance

We measure the pure GPU kernel execution times in order to compare head-to-head how our method performs in terms of the kernel-level performance against the FFT-based convolution methods in cuDNN. cuDNN is generally considered as a library that is deeply optimized by NVIDIA and provide some state-of-the-art and fastest convolution implementations. We vary kernel sizes, and batch sizes to analyze strengths and weaknesses for these algorithms in the parameter space. We organize

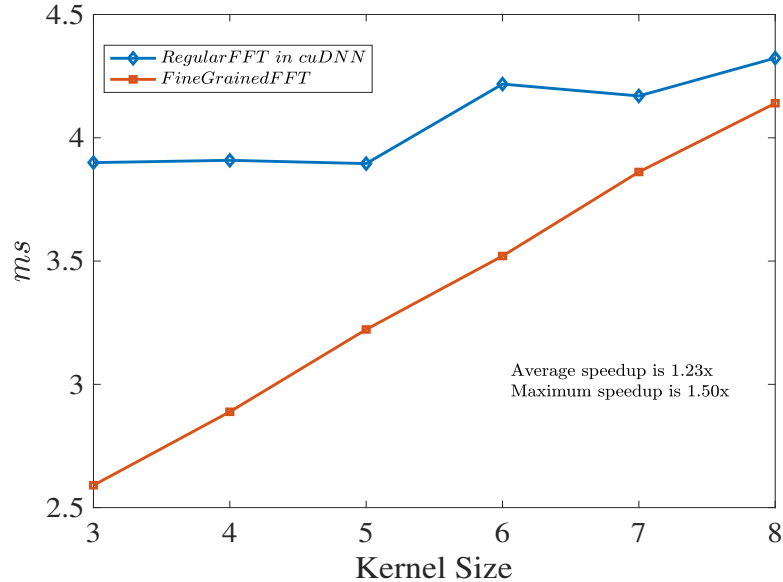


Figure 4.9: RegularFFT and FineGrainedFFT performance comparison as the kernel size varies from 3 to 8 on ILSVRC2017 benchmark

these parameters into a 2-tuple (U, N) , and we assign a set of values to the other parameters (K, C, H) that is commonly used in benchmarking convolution performance. The experiment is then categorized into two groups. Each group fixes the value of one parameter and varies the other one. Thus, we can study how this parameter impact the overall performance of the algorithm. For the kernel-wise comparison, the total execution time does not include the first call to cuFFT library since it has significant context initialization cost. To exclude it from the reported timing, a warmup call is performed to isolate the cost. Please note that the performance of our method is dependent on the parameters of convolution. However it is insensitive to the value of inputs and weights. Due to the performance’s insensitivity to input, the performance we observe at the kernel level is going to be consistent with that with real-world data.

In Figure 4.9, we compare the fine-grained FFT algorithm with the regular FFT algorithm from cuDNN on a synthetic benchmark and the 2017 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) object localization benchmark. We use random input data and kernels from $[-1, 1]$ for the synthetic benchmark. The execution time of regular FFT convolution tends to be constant and insensitive to the kernel size,

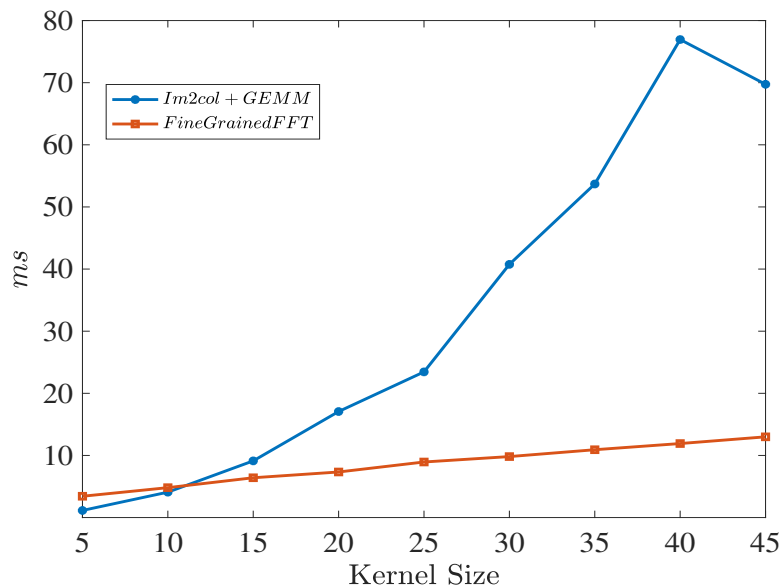


Figure 4.10: Im2col+GEMM and FineGrainedFFT performance comparison as the kernel size varies from 5 to 45 on synthetic benchmark

because it zero-pads the kernel to be the same size as the input image and kernel size has nearly no effect on the performance. In contrast, the performance of our method decreases with the kernel size since our algorithm is based on the padded matrix. Although the matrix is not unrolled, the algorithmic complexity is still dependent on the kernel size. Figures 4.8 and 4.9 have similar trend. Our fine-grained FFT convolution maintains its high performance when the kernel size is small. It is shown that our algorithm is insensitive to the value of inputs and kernels.

In figure 4.10, we also compare the fine-grained FFT algorithm against the im2col+GEMM algorithm from cuDNN on the synthetic benchmark. Although the execution time for both approaches increases as the kernel size increases from 5 to 45, the im2col+GEMM approach increases more rapidly. Our approach outperforms the im2col+GEMM one when the kernel size is greater than 10, because the fully unrolled matrix grows quadratically with the kernel size for the im2col+GEMM approach. In contrast, our approach does not fully unroll the input. In addition, the kernel and input matrix multiplication performed by FFTs has lower algorithmic complexity. Compared

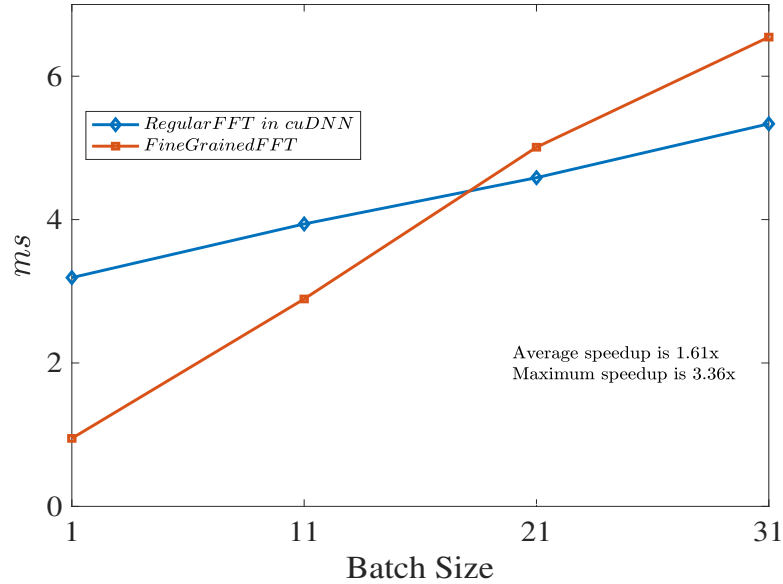


Figure 4.11: RegularFFT and FineGrainedFFT performance comparison as the batch size varies from 1 to 31 while other parameters are fixed.

with the im2col+GEMM algorithm, our method has better performance when the kernel size is large. However, when the kernel size is small, im2col+GEMM has better performance, because the unrolled matrix is small and the high performance of matrix multiplication routine in cuDNN outweighs the saving of algorithmic complexity of fine-grained FFT method. For the im2col+GEMM approach, execution time in kernel size 45 is smaller than that of 40, it is likely due to the matrix dimensions are within favorable regime and this particular dimensions often achieve the best achievable performance.

We then evaluate the batch-mode performance. Figure 4.11 shows the performance comparison with varying batch sizes. The two curves cross the intersection point around 20, and the fine-grained FFT outperforms the regular FFT for the smaller batch size. As presented in Table 4.1, batch size has large negative impact on the element-wise multiplication that it may offset the performance gain in FFT as the batch size increases. In both Figures 4.8 and 4.11, two curves have an intersection point indicating that the fine-grained FFT convolution excels with small kernels and batches, and for a small region of the parameter space, the regularFFT approach is better.

Conv. Layers	L1	L2	L3	L4	L5
Network1	(3, 10)	(3, 5)	(3, 8)	(3, 7)	(3, 10)
Network2	(3, 5)	(4, 10)	(3, 5)	(5, 5)	(3, 5)
Network3	(3, 10)	(3, 8)	(5, 5)	(3, 10)	(3, 5)

Table 4.4: Layer configurations for the three synthetic CNNs. Their performance evaluation is shown in Figure 4.12. Each element in the table indicates (U, K) .

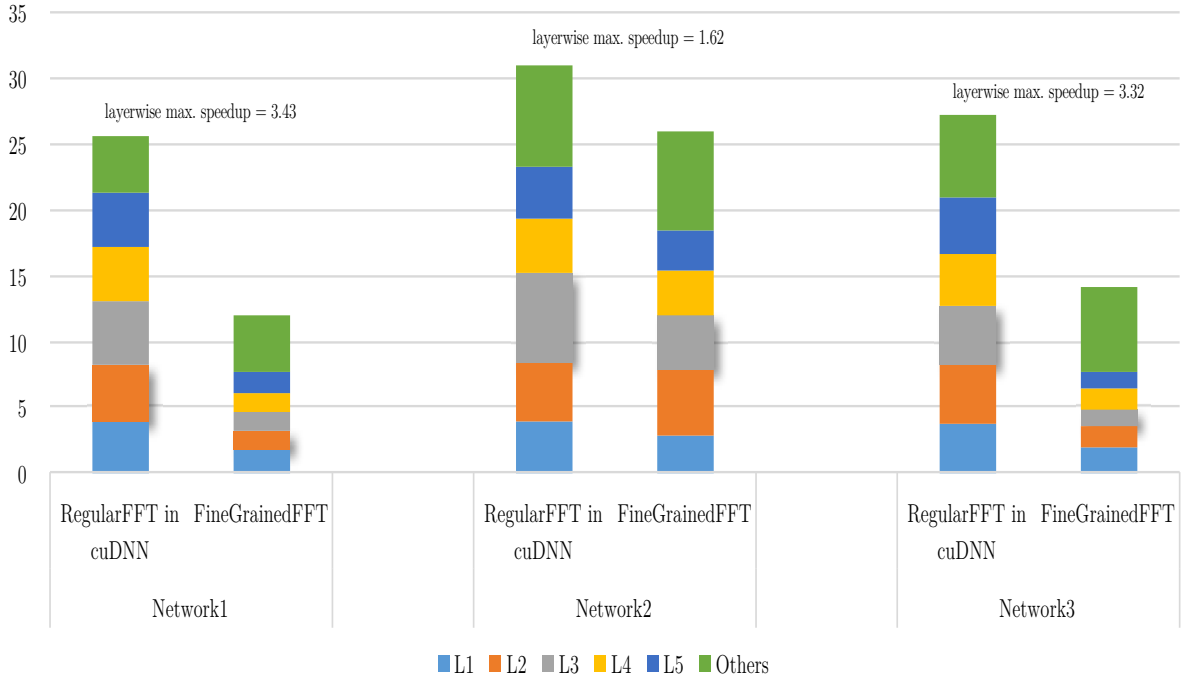


Figure 4.12: Layer-wise performance benchmark for the networks composed by five convolutional layers. Others in the legend represents pooling, ReLU and fully connected layers. Conv. layers with maximum speedup are highlighted as shaded rectangles. The average speedup for the three networks is $1.74\times$.

4.4.3 Performance in Application

Caffe is one of the most popular frameworks that people use to develop deep-learning applications. In this experiment, we replace the convolution implementation in Caffe with our method, compose several CNNs with Caffe, and compare the performance before/after the replacement.

We compose three CNNs by five convolutional layers with parameter configurations listed in Table 4.4. The CNNs are inserted pooling and rectified linear units layers, and the last layer is a fully connected layer for prediction with 10 outputs. The

inputs to these CNNs are $128 \times 128 \times 3$, $254 \times 254 \times 3$ and $254 \times 254 \times 3$ images with batch sizes of 5, 10 and 1, respectively. As it is shown in Figure 4.12, our fine-grained-FFT convolution outperforms the regular-FFT one in all configurations for one iteration of CNN inference. Specifically, it achieves speedups of $2.12\times$, $1.19\times$ and $1.92\times$ over the regular-FFT convolution, respectively. Additionally, our method perform faster for the layer-wise comparison, except for L2 in Network2. Unsurprisingly, our method has larger speedups on small kernels and small batch sizes.

4.4.4 Performance Analysis

In order to empirically explain the performance gain of our algorithm, we use nvprof to profile the GPU kernels and compare the FFTs and element-wise multiplication execution time for each algorithm side by side in Table 4.5. This performance profiling provides a detailed performance breakdown and enable us to see which steps contribute most to the performance gain.

KernelSize	FFT_r	FFT_f	MM_r	MM_f	Speedup
3	0.732	0.784	3.272	1.510	$1.74\times$
4	0.730	0.681	3.267	1.812	$1.60\times$
5	0.725	0.770	3.271	2.200	$1.34\times$
6	0.732	0.772	3.283	2.599	$1.19\times$
7	0.728	0.648	3.267	2.848	$1.14\times$
8	0.724	0.787	3.257	3.085	$1.02\times$
9	0.691	0.764	3.249	3.307	$0.96\times$

Table 4.5: Profiling results with varying kernel size. Subscripts of r, f indicate RegularFFT and FineGrainedFFT convolutions. FFT and MM represent the FFTs and element-wise multiplication execution time. (unit ms)

The FFT_r and MM_r stages have almost constant performance as kernel size increases because the kernel zero-pads to be the same size as the input feature maps, thus the amount of computations does not change. In contrast, the element-wise multiplication for the fine-grained-FFT convolution MM_f grows as the kernel size increases because our method is dependent on the im2col process; as kernel size increases the unrolled matrix becomes larger. FFT_f also tends to be a constant since the FFTs size

is a power of two depending on the input size. In this case, it is 512. The matrix multiplication MM_r implementation is in fact batched matrix multiplications and perform transpositions to prepare tensors for matrix multiplications, which incurs extra operations. Thus MM_r is larger than MM_f except the last row in the table. Additionally, FFT_r and FFT_f almost have the same values. For 3×3 kernel, the speedup is $1.74\times$, which also means fine grained FFT convolution is 33% faster than the regular FFT one.

4.5 Summary

In this chapter, we propose a fine-grained FFT convolution and it achieves competitive performance when the kernel and batch size is small in the parameter space. The contribution of this work is threefold. First, we identify the doubly block hankel matrices of the unrolled input feature maps in the im2col process. Second, we propose an implicit matrix multiplication in the Fourier domain, which can reduce memory requirements. Finally, the fine-grained FFT convolution is implemented. In contrast to the regular convolution that takes 2D FFT over the input feature map, our approach has fine FFT granularity.

Chapter 5

SCALABLE TOP-K QUERY PROCESSING

5.1 Overview

Top-K query processing is one of the fundamental and the most performance-deciding components in Web search engines. A number of techniques such as dynamic pruning have been proposed to reduce the query processing time on CPU. However, it has become increasingly difficult to further improve Top-K query processing's efficiency without hurting its effectiveness. On the other hand, Graphic Processing Unit (GPU), a powerful computing accelerator on almost every computer today, is barely tapped in Web search engines. The biggest challenge to accelerate top-K query processing on GPU is that the parallel nature of execution model of GPU prevents many CPU top-K query processing optimizations from being directly ported to GPU. GPU with hundreds of cores is ideal for applications with massive parallelism, which is not readily available in existing CPU-oriented top-K query implementations.

This work exploits the GPU computation power for top-K query processing. In particular, we propose a new domain-specific parallelization framework to utilize GPU to parallelize it. The proposed framework is general enough for both disjunctive and conjunctive query processing modes. Experiments on TREC collections show that our proposed GPU top-K query processing framework is able to improve the query processing time by a factor of 7 when compared with state-of-the-art dynamic pruning methods for the disjunctive mode and by a factor of 6 when compared with the conjunctive mode. Our results show that our GPU top-K query processing framework is faster than previously known GPU baseline method. In particular, our framework is shown to be more scalable and efficient than the CPU and GPU baselines when K is large.

5.2 Introduction

Large-scale Information Retrieval (IR) systems, such as Web search engines, rely on fast response and high throughput to deal with rapid growing number of queries and web pages. The efficiency of a search engine can directly affect its revenue as well as users' search experience [64]. Given a query, an IR system needs to compute the relevance score for each document based on an underlying retrieval function, and then returns top K documents with the highest relevant scores. This process is known as top-K query processing. Although most Web search engines adopt a multi-stage distributed architecture to process queries [5, 76], top-K query processing on a single node is still the first step to quickly identify a set of promising documents that need to be re-ranked with more complicated ranking mechanisms. Clearly, reducing top-K query processing time is a crucial step to improve search efficiency.

The most basic query processing strategy is exhaustive query processing using *disjunctive* (OR) mode, which evaluates all documents containing at least one query term and then ranks them based on their relevance scores. Although this strategy is simple, the computational cost would be quite high, in particular when an IR system has to deal with hundreds of millions of documents. An alternative is to process queries using *conjunctive* (AND) mode, which means that relevance scores are computed only for documents that contain all the query terms. This significantly decreases the number of evaluated documents and thus reduces the query processing time. However, it hurts the retrieval effectiveness significantly since many relevant documents do not contain all query terms [53, 79]. Various dynamic pruning methods [8, 73, 22] have been proposed to reduce the query processing time for the disjunctive mode without hurting the retrieval effectiveness. The main idea is to avoid evaluating documents which are unlikely to make to the top-K search results. Although these methods can improve the search efficiency when the number of returned documents (i.e., K) is small, they are not scalable for larger K 's [79]. This is probably because the overhead of the dynamic pruning methods (such as pre-computing necessary statistics and storing them on the disk) increases as K gets larger. In fact, it becomes increasingly difficult to

further improve the efficiency of top-K query processing without any degradation of effectiveness in the search results. Recently, researchers have started to look into how to sacrifice effectiveness for the sake of further reducing the query processing time [79, 72].

Almost all of today's top-K query processing implementations were developed on and tuned for CPU. A significant source for computation power in today's computers is unused. Most of today's computers have not only CPU but also GPU, i.e., Graphics Processing Unit. GPU is a powerful platform that has been successfully used to accelerate various computer-intensive applications. Despite its great potential, GPU has not been fully utilized to improve the search efficiency. This is largely because porting top-K query processing to GPU is hard. The first challenge is how to effectively utilize GPU's massive parallelism in top-K query processing. The number of computing cores on a GPU is large, e.g., thousands cores in most powerful GPU models, which makes it possible for massive parallelism. However, top-K query processing is not a task that can be easily massively parallelized. One naive way is to set the number of threads the same as the number of query terms. However, the degree of parallelism here, i.e., the number of query terms, is much smaller than what the GPUs can do, leading to underutilization of the GPUs. The second challenge is how to structure the computations and the data transfers involved in the top-K query processing so that we can effectively leverage the programming model offered by the GPU. In particular, GPU runs most efficiently when threads execute the same workload at the same time, which is called SIMT (Single-Instruction-Multiple-Thread). SIMT requires very different way to express computation workload than on CPU. It means that the computations in CPU-oriented top-K query processing need to be structured in a way that all threads that are executed at the same time better to perform the same task. This is not trivial because many steps in the top-K query processing are adaptive and it requires careful designs when we need to massive parallelize each step. Ding et al. [21] tried to leverage the GPU for query processing, but they focused on query processing for a small value of K (i.e., $K=10$). It remains unclear whether there is a general strategy for GPU-based

top-K query processing and how well it can scale with K .

In this work, we propose a novel framework of exploiting the parallelism of top-K query processing on GPUs. Our framework presents the same interface as existing top-K query processing implementations, that is, queries are submitted to a CPU. However, for each query, the CPU transfers the compressed inverted indexes related to the query to a GPU, and the GPU then evaluates documents and returns top-K results back to the CPU. The processing time of a query consists of the query processing time spent on the GPU as well as the data transfer time. The main innovation of our framework is that we leverage the data-parallel programming model provided by the GPU to speed up the process of document evaluation. Generally speaking, the document evaluation process consists of three steps: *index decompression*, *score calculation* and *top-K selection*. Since these three steps require different types of computations, we have designed different strategies, such as blocked scan, double-level binary search, bucket selection, to parallelize each step. Unlike the previous study [21], our framework is general enough to process queries in both disjunctive and conjunctive modes.

Experiments are conducted over multiple TREC Web collections. Results show that the proposed GPU top-K query processing framework can significantly improve the efficiency compared with the CPU baseline method in both disjunctive and conjunctive mode over all collections. When compared with the exhaustive query processing on CPU, the average speedup is around 33 for the disjunctive mode and 6 for the conjunctive mode. The GPU-based methods are more efficient than the state of the art dynamic pruning methods. It can also outperform the previously proposed GPU-based method [21], in particular, when K is larger. Moreover, empirical results consistently show that the GPU-based methods are scalable with respect to K , i.e., the speedup remains the same as K gets larger. Finally, it is interesting to note that, with the proposed GPU optimization methods, the processing time of the disjunctive mode is comparable to that of the conjunctive mode, which means that we can significantly improve the efficiency without any sacrifice in terms of the effectiveness.

5.3 Related Work

Improving search efficiency has been an active research topic since the beginning of the IR field. Commonly used strategies include index compression[81], caching[11], dynamic pruning[8, 73, 22], distributed computing[6], query processing on multicore architecture [71] etc.

Although GPU is a powerful platform used to accelerate computing-intensive applications, not many studies focus on top-K query processing on GPUs. There are a few studies that used GPUs to improve the efficiency for applications related to the top-K query processing, such as list intersection [78, 4] and relational operations [32]. However, they only solve one step involved in the top-K query processing, and none of them provided a complete solution to top-K query processing.

Ding et. al. [21] was the first and probably the only study done on using GPUs for top-K query processing so far. They mainly focused on conjunctive query processing mode and the results are evaluated only when K is set to 10. On the contrary, we propose a general framework that can process queries in both disjunctive and conjunctive modes with scalability. We evaluated the proposed methods on multiple values of K , and found that the proposed framework is scalable and can still keep large speedup even when K is large. Finally, another key difference is that we do not assume that inverted lists are available in GPU global memory when processing queries on the GPUs, which is an assumption made in the previous study [21]. The assumption might give unrealistic advantage to GPU-based implementations. Instead, when measuring the performance, the query processing time includes the time spent on transferring data between CPU and GPU memories. This transfer is often considered to be one of the bottlenecks when applying GPU to accelerate applications, but, as shown in this work, even with this overhead the proposed GPU framework can still achieve significant speedup.

The contribution of this work can be summarized as follows. First, we propose a novel framework that can fully exploit the massive parallelism power of GPU to speedup the top-K query processing time. The framework is general enough for both

disjunctive and conjunctive modes. Second, experiment results show that the proposed GPU framework is more efficient and scalable than the state-of-the-art CPU and GPU top-K query processing methods. Finally, with the GPU optimization, for the first time, the query processing time for the disjunctive mode is comparable to that for the conjunctive mode, making it possible to improve the efficiency significantly without hurting the search effectiveness.

5.4 Top-K Query Processing Background

Web search engines use inverted indexes to facilitate the search process. For each term in the collection, an inverted index was built to store the information about the occurrences of the term in the documents. An inverted list consists of a list of postings, where each posting contains a document ID and the occurrences of the term in the corresponding document. Indexes are often stored in a highly compressed format. The compression not only reduces the total size of index, but also improves efficiency by decreasing the number of disk reads. Since the indexes are compressed, the first step of the top-K query processing is often to decompress the indexes to get the term statistics.

With the decompressed statistics, search engines need to traverse inverted indexes to compute relevance scores for all documents based on an underlying retrieval function. There are two commonly used index traversal methods: Term-At-A-Time (TAAT) [10]. and Document-AT-A-Time (DAAT) [8]. TAAT sequentially processes one query term at a time. It goes through the inverted list of a term and accumulates the partial document scores contributed by the term. The partial scores are stored in an accumulator, and will later be accumulated to compute the final document scores. On the contrary, DAAT processes one document at a time. It goes through the inverted lists of all query terms in parallel. A document needs to be fully evaluated before moving on to the next one. Since DAAT requires the synchronization among posting lists, it is not suitable for the highly parallel architecture of the GPUs due to

data dependency. Therefore, in the proposed GPU implementation, we use the TAAT query processing strategy.

Given a query, it can be processed either using *disjunctive* (OR) mode or *conjunctive* (AND) mode. In the disjunctive mode, we compute the scores for all documents with at least one query term. In the conjunctive mode, we compute the scores only for documents with all query terms. The conjunctive mode is often considered more efficient and less effective, since it evaluates fewer documents and some relevant documents may not contain all query terms [79]. In this work, we develop a general framework that can process queries in both modes.

5.5 GPU-Based Top-K Query Processing

Our GPU-based top-K query processing framework works as follows. Given a query, CPU sends the query as well as the inverted indexes of all the query terms to GPU. The GPU then evaluates documents based on the query and the inverted indexes, and returns top-K ranked documents to the CPU. Note that the query processing time here includes the time spent on transferring the indexes, the time spent on document evaluation, and the time spent on returning the search results.

Unlike the previous study where the entire inverted indexes are assumed to be kept in the GPU memory, our framework keeps only simple global statistics such as document lengths and IDF values in the GPU memory. The main reason of our design is that GPU memory has limited size and it may not be able to hold the entire inverted indexes for large data collections. Therefore, all the inverted indexes are kept on the CPU side, and only those related to the query will be transferred to the GPU. As we will show in the experiments, even with the overhead of the data transfer, the proposed GPU framework is still able to improve the search efficiency significantly.

The transfer of data between GPU and CPU is pretty straightforward. The main challenge, also the main technical contribution of our framework, is the parallel implementation of document evaluation for the GPU-based top-K query processing. During the document evaluation, the system first needs to read the compressed indexes

and decompress them to get the posting information. After that, we need to traverse the indexes and compute the relevance scores. As discussed earlier, we use the TAAT method for index traversal. In particular, a large array is allocated to record the relevance score for each document with respect to a query term, where the size of the array is the total number of documents for the query term. When processing a query with TAAT, the GPU would first go through each posting list and compute the partial relevance score of a document with respect to the corresponding query term. After that, we can compute the relevance scores of all documents by combining the partial scores in all the posting lists of a query. To do this, list operations (either intersection or union) need to be performed. Finally, we need to select top-K ranked documents from the list based on their scores.

Clearly, when implementing the query processing with GPUs, we can divide document evaluation into the following three steps: (1) *index decomposition*, which decompresses the indexes related to the query terms; (2) *score calculation*, which calculates the partial relevance scores of documents for the posting list of each query term and then combines scores from multiple posting lists through different list operations (i.e., list intersection for conjunctive query processing mode and list union for disjunctive query processing mode); and (3) *top-K selection*, which goes through the final list and selects documents with top-K highest scores. We describe how to parallelize each component for GPUs in the following subsections.

5.5.1 Parallel Index Decompression

As mentioned in the previous section, given a query, the CPU transfers the inverted indexes of the query terms to the GPU. Since the indexes are compressed, the first step is to decompress the indexes.

One possible solution to parallelize index decomposition is to decompress the inverted index for all the terms in parallel. Since the number of terms in a query is not big, such parallelism would under-utilize the GPU. Recall that each term has an inverted index with a list of postings, and the posting list of a common term could

contain billions of postings. Thus, a more sensible solution could be to parallelize the decompression of the posting list of a common term. We now provide more details on how we tackle this challenge.

PForDelta is a commonly used index compression method for IR systems [82]. Like all the other index compression methods, PForDelta does not directly store the document IDs in each posting list because the document IDs can be fairly large numbers. Instead, it stores the differences between the sorted document IDs in each posting list, and these gaps are then compressed. PForDelta first splits the data into blocks and decompresses one block at a time. The size of a block needs to be a multiple of 32, and we set it to 64 in this work. For each block, PForDelta chooses an integer b so that a certain percentage (e.g., 90%) of the gaps in a list can fit into a fixed length field with b bit. The remaining gaps, i.e., those are larger than 2^b , are referred to as exceptions. PForDelta can be tuned by choosing different thresholds for the number of exceptions allowed. Because exceptions and non-exceptions are compressed using different numbers of bits, it is difficult to decompress both of them simultaneously on the GPU in one CUDA kernel invocation. In order to exploit more parallelism, we set the number of exceptions allowed to zero. In other words, for each block, we choose the value of b such that all the gaps in the block are smaller than 2^b . As a result, all the information in the indexes are compressed using the same strategy. It is expected that the compressed index size would increase because of this new increased value of b . Our result shows the compressed size of Gov2 collection increases from 8.2GB to 11GB. However, the major benefit of using this variant is to eliminate the kernel invocation overhead of exceptions decompression as well as provide a uniformly parallel decompression scheme for the whole block. It is worth pointing out the decompression technique CPU query processing used in this work is PForDelta since it is more efficient than our parallel decompression method.

Figure 5.1 illustrates the basic idea of index compression method with a simplified example. We assume that the term t occurs in multiple documents whose IDs are $dID_1, dID_2, \dots, dID_{64}$. Instead of storing these IDs directly in the posting list, we store

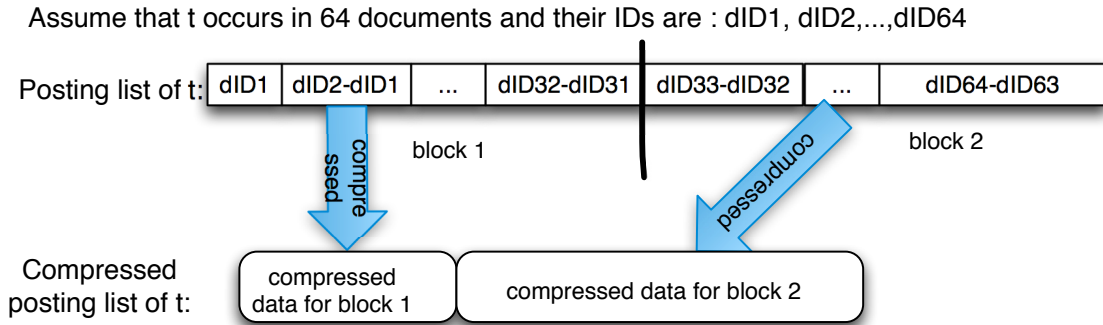


Figure 5.1: An example of index compression

the gaps, i.e., the differences between the sorted document IDs such as $dID_2 - dID_1$. Note that other information (such as term frequency and term position) also needs to be stored together with document ID gaps. We want to keep the example simple, so did not show those information here. The posting list can be divided into blocks (32 postings per block in this example), and each block can then be decompressed using different values of b to make sure that all of the gaps in the block are smaller than 2^b . Since the sizes of the compressed data blocks could be different, when storing the compressed postings for each block, we also need to store the value of b and the length of the compressed block.

When decompressing the indexes, we need to recover the posting lists including the document ID and term frequency for each document and term pair. Since an inverted list needs to be split into blocks when using PForDelta for compression, we can first look at the main computations involved in each block and then discuss how to process the entire list. The main computations involved in each block include:

- *block address calculation*, to compute the starting address of each compressed data block;
- *block decompression*, to read and decode a block based on the block address;
- *document ID recovery*, to read the document ID gaps from the decompressed blocks for the document ID gaps, and recover the original document IDs based

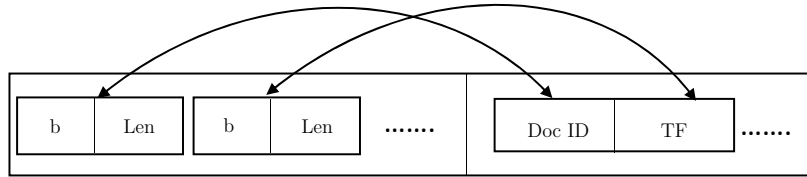


Figure 5.2: The layout of compressed data for each block in compressed postinglist on the ID gaps.

5.5.1.1 Block Address Calculation.

The first step of the decompression is to read each compressed block. Since the length of each block is not fixed, we need to use the lengths of compressed blocks to recover the address for each block. Figure 5.2 zooms in the compressed data for block in Figure 5.1, and it shows the internal layout for each compressed data block in the compressed postinglist. The respective values of b and the length of each compressed block are denoted as b and Len in Figure 5.2. If we want to read the compressed blocks in parallel, we have to figure out how to compute the starting addresses of all the blocks in parallel.

Take Figure 5.1 as an example, the address of the second compressed block can be computed by summing up the address of the first block and the length of the first block. Similarly, the address of the third compressed block, if there is any, can be computed by adding the address of the second block with the length of the second block. In fact, this process is essentially to compute the prefix sum for all the lengths of compressed blocks, and can be parallelized using parallel scan. Parallel scan is a widely used parallel operation on GPU and can be used in applications such as sorting, stream compaction, building histogram, etc. [31, 65]. When computing the starting address of a block, we do not use the length of the current block, so this kind of prefix

sum is exclusive. In this work, we apply an exclusive parallel scan to compute the starting address of each compressed block of data. Regarding the GPU configuration, we set the thread block size to 1,024, and each thread is assigned to process an integer, which contains the information about two postings.

5.5.1.2 Block Decompression.

After reading all the compressed blocks, the next step is to decompress these blocks in parallel. This step is pretty straightforward. Since an inverted list is split into blocks and each block contains the information about 64 postings, we launch a block of 64 threads to decompress each compressed data block and the total number of threads is equal to the number of documents in the posting list, and thus, the compressed posting can be decompressed simultaneously.

5.5.1.3 Document ID Recovery.

After decompressing the data blocks, we can get the document ID gaps for each posting list. The next step is to recover the original document IDs. For example, as shown in Figure 5.1, dID_1 and $dID_2 - dID_1$ are stored in the posting list and we can get the original document ID of the second document, i.e., dID_2 , by adding the gaps up. How do we parallelize this process to recover a large number of document IDs in the same time? This can still be solved using parallel scan since the computation is inclusive prefix sum. However, since the posting lists could be very long, it may require multiple levels of recursions to finish the scan, adding extra scan kernel invocations. To improve the efficiency, we propose a segment-based parallel scan. The main idea is to split each posting list into segments and apply an inclusive parallel scan on each segment. For each inverted list, we build an array called *FirstID* to store the original ID for the first document in each segment, and the array elements are then later used together with the segment-based parallel scan results to recover the original ID for all the documents. We also tune the thread block size and set it to 128 based on our preliminary results, and each thread processes 8 postings to better cover the global

memory latency for each thread [52]. So, the size of a segment is set to 1,024, meaning each segment contains 1024 postings. Our preliminary results show that the proposed segment-based parallel scan is more efficient than the original parallel scan, achieving a speed up of 1.4 for an array with 2^{24} elements.

5.5.2 Parallel Score Calculation

After decompressed the indexes, the system can then traverse the indexes to compute the relevance scores for all the documents. The relevance scores of a document with respect to a query is often computed by summing up the partial relevance score of a document with respect to a term for all the query terms [62]. Thus, three main computations involved in this step include:

- *partial score calculation*, which computes partial relevance scores of a document for matching each of the query term based on an underlying retrieval function;
- *score accumulation*, which accumulates all the partial scores of a document with respect to each query term and computes the final relevance score for the document.
- *unique document filtering*, which filters out duplicate documents between posting lists, and leave unique documents with final score for top-K selection.

In the *partial score calculation* step, for each query term, we need to go through its posting list, calculate the partial scores of each document on the list and record them in large score arrays. They are allocated in GPU global memory with size of the total number of documents for each term, as we already mentioned in the introduction of our general framework. Auxiliary information such as total number of documents for each term is saved as part of the indexes. Since we need to compute the partial score for each term and its associated document, we can parallelize this step by allocating one GPU thread for each document in the term utilizing BM25 [62] function. Specifically, it is a combination of inverse document frequency, term frequency in the document, the length of the document and average document length in the collection.

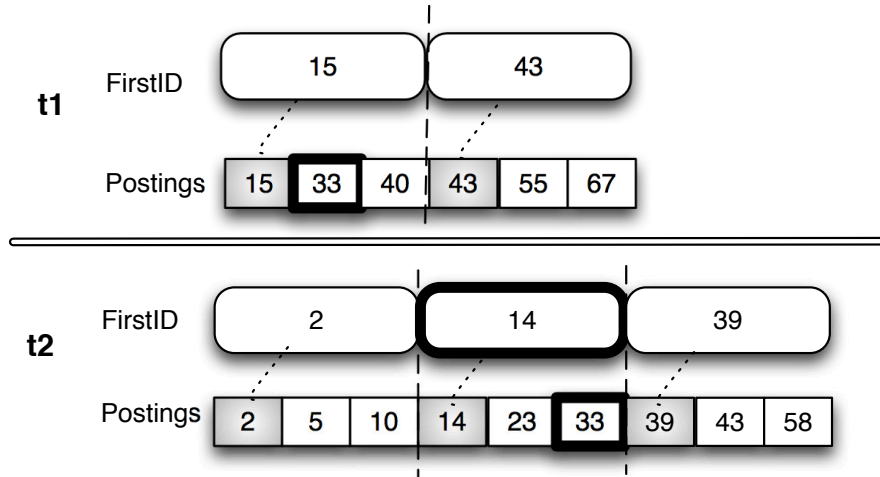


Figure 5.3: An example of two-level search

Here, we have completed partial score calculation for each document with respect to the query, our next step is to accumulate partial score and compute the final score. When computing the final score of a document, we need to find all the posting lists that contain the document and sum up the partial scores. Specifically, given a document (i.e., posting) in an inverted list, we need to locate the document in the other inverted lists so that the partial scores of this document could be accumulated. The main computation here is to look up the document in the posting lists. To speed up the process, we propose to leverage the *FirstID* arrays discussed in Section 5.5.1.3. Recall that we split an inverted index into segments, and *FirstID* stores the original ID of the first document in each segment. The *FirstID* array are preprocessed offline, as a part of auxiliary data structure to the indexes. Its space overhead is acceptable, e.g., *FirstID* array in GOV2 is 345MB comparing with 11.5GB for the compressed posting lists. The ratio between them is 3%. Thus, given a document, we employ a two-level binary search to locate its location in the posting lists. We first use *FirstID* arrays to narrow down the search space, and then use the document IDs to identify the exact location. To parallelize the above process, we allocate a thread for each document, and then conduct two-level parallel binary searches to locate the positions of the posting

in all the inverted lists and compute the final score. In this step, we utilize a predicate array to save boolean variables, if a duplicate document is found during two-level search, the corresponding location of predicate array is marked as TRUE. Before the parallelization process, we sort the decompressed posting lists by their lengths in an ascending order, which minimizes the number of allocated thread blocks and further reduces the overhead on GPU.

Figure 5.3 illustrates an example. Assume that a query has two terms t_1 and t_2 and each segment contains 3 postings in this example. t_1 has a shorter posting list than t_2 , so we would first go through the postings of t_1 to calculate the final document scores for each of them. When computing the final score for document 33, instead of searching a match on the posting list of t_2 , we first search over the *FirstID* array for t_2 to identify the corresponding segment on the posting list of t_2 and then search the elements in the segments.

There might be duplicate documents between posting lists. Suppose a query contains term A and B, both posting lists of A and B have duplicate document I (called I_A and I_B). In the previous score accumulation step, the score array of I_B contains the total score for the query while I_A contains partial score. Thus, it requires to filter out duplicate document with partial score. Otherwise, duplicate documents might be selected into tok-K results. In this unique document filtering step, we adopt the implementation in [2]. Its key idea is to use shared memory atomics to filter out duplicate documents by the Predicate array mentioned above.

Note that the above parallel method can be applied to both disjunctive and conjunctive query modes. For disjunctive mode, we need to allocate threads for documents that occur in the inverted lists of the query terms, i.e., those containing at least one query term. For conjunctive mode, we need to allocate threads to only documents that occur in the shortest posting list. To establish a connection among these three steps, we present the following pseudocode for conjunctive mode. The disjunctive mode can be established similarly. In Algorithm 1, N is the number of query terms. PRED represents the predicate array mentioned in two-level search. It is also used in

document filtering step. $DocID_i$ represents the corresponding decompressed posting lists for each term. They are sorted by their length in ascending order.

Algorithm 1: Parallel Score Calculation

```

Function{ScoreCalculation}{ $DocID, Score, FirstID, PRED, N$ }
  PartialScoreCalculation{ $DocID, Score$ }
  for  $i \in \{2, \dots, N\}$  do
    ScoreAccumulation{ $DocID_1, DocID_i, Score, FirstID, PRED$ }
  end for
  DocumentFiltering{ $DocID, Score, PRED$ }
EndFunction

```

5.5.3 Parallel Top-K Selection

We have discussed how to leverage GPU to parallel decompress indexes and compute the final relevance scores for documents. This section focuses on how to select top-K ranked documents using GPU.

The simplest strategy is to sort all documents based on their scores and pick the top-K ranked documents. However, this might unnecessarily waste a lot of computational power because we do not care about the ranking of a document if it does not make it into the top-K and K is often much smaller than the total number of documents in the collection. To speed up this process, we propose a method based on bucket sorting. The main idea is to first distribute documents into a number of buckets based on their relevance scores, select a minimum number of buckets that can cover the top-K ranked documents, and identify top-K ranked documents from the selected promising buckets.

The first step is to divide documents into buckets based on the scores and then select a minimum number of buckets that can cover top-K documents. Assuming there are a number of buckets, each of them corresponds to a range of relevance scores, and the documents in a bucket should be within the corresponding score range of the bucket. Therefore, the relevance score of a document decides which bucket it would be put in.

When deciding on the score range for each bucket, we first use the collection statistics to compute the maximal and minimal values for the relevance score and then divide the score range evenly based on the number of buckets, e.g., Based on the statistics about the GOV2 collection and BM25 function, we choose max value to be 74, and min value is 0. Thus, let B denotes the number of buckets, max and min denote that maximum and minimum of the relevance scores, and $score[i]$ denotes the relevance score of document i . We can determine the bucket number for the document (i.e., $bucket[i]$) as: $bucket[i] = B - \lfloor \frac{B}{max-min} \cdot (score[i] - min) \rfloor$

During the process of documents distribution, we also maintain an array in GPU global memory to save the bucket number for each document. After assigning a document to its corresponding bucket based on the above equation, we need to count the number of documents in each bucket. We now explain how to parallelize this step. Each bucket maintains a counter to record the number of documents in the bucket. We allocate one thread for each document. So when we assign a document to a bucket, the corresponding thread needs to *atomically* increase the counter of the corresponding bucket by 1. When multiple threads need to add the value to the same bucket, we may encounter the problem of collision and need to make sure the operation to be atomic. It is well known that atomic operations on global memory in GPU is computationally expensive, especially in the case of large collision volume, and atomic operations in the shared memory is faster than in the global memory. Therefore, we have adopted a method from the previous study [67] to simulate atomic add in the shared memory. Our preliminary results show that this method can reduce the collision rate of atomic operations and achieve a speed up of 2. After counting the number of documents for each bucket, we can perform a serial cumulative sum and figure out how many buckets include the top-K documents.

With the identified buckets, we can then sort all the candidate documents with any existing sorting algorithms. We used radix sorting algorithms [50] in this work, since it is considered as one of the fastest sorting algorithms on GPU, and Thrust library [35] in CUDA includes its implementation. Radix sorting is a non-comparison

based sorting algorithm, which considers one bit from each key, and partitions the unsorted array elements so that all elements with a 0 in that bit precede those with 1 in that bit. When GPU finishes selecting top-K documents, it returns the retrieval results back to CPU.

5.6 Experiments

To evaluate the efficiency of the proposed GPU-based top-K query processing framework, first, we compare the proposed parallel GPU top-K query processing methods with the CPU top-K query processing methods for the *exhaustive* evaluation, where all the candidate documents are evaluated and ranked. The methods are compared in both disjunctive and conjunctive modes. Second, we compare the proposed GPU methods with several state of the art top-K query processing methods, which includes dynamic pruning methods maxScore [73] and Block-Max WAND [22] as well as the previously proposed GPU top-K query processing method for both conjunctive and disjunctive modes [21]. Additionally, we conduct more analysis to further understand the proposed GPU methods.

The proposed GPU framework is implemented on Nvidia Tesla C2075 with 448 CUDA cores. All CPU query processing methods are evaluated on a single core of Intel Core i7 CPU. All the methods use the same indexes, which are kept in the CPU memory. Relevance scores are computed based on Okapi BM25 [62] in our experiments, but the proposed GPU framework can work with any retrieval functions. The number of buckets (i.e., B in Section 5.5.3) is set to 32 because the size of a warp in GPU is 32 and it is easier to implement the atomic operation in shared memory when setting B to the same value as the number of threads in a warp. The code of our proposed methods will be made available at GitHub for other researchers to use and study in the future.

Experiments are conducted over multiple TREC collections. The first three were used in the TREC 2004-2006 Terabyte tracks, and their document collection (i.e., GOV2) consists of 25 millions of webpages. The data sets are denoted as *TB04*, *TB05*

Table 5.1: Performance comparison on exhaustive evaluation (ms)

(a) Disjunctive (OR) mode (K=1,000)

	<i>TB04</i>	<i>TB05</i>	<i>TB06</i>	<i>Web09</i>	<i>Web10</i>	<i>Web11</i>	<i>Web12</i>
CPU-OR	683.99	577.70	545.55	1038.70	752.26	1557.03	1054.51
GPU-OR (Speedup)	21.09 (32.4)	17.94 (32.2)	16.40 (33.3)	31.08 (33.4)	21.82 (34.5)	43.45 (35.8)	31.15 (33.9)

(b) Conjunctive (AND) mode (K=100)

	<i>TB04</i>	<i>TB05</i>	<i>TB06</i>	<i>Web09</i>	<i>Web10</i>	<i>Web11</i>	<i>Web12</i>
CPU-AND	73.53	43.16	43.81	172.78	106.72	103.55	172.76
GPU-AND (Speedup)	12.90 (5.7)	11.12 (3.9)	10.40 (4.2)	22.80 (7.6)	15.80 (6.8)	25.50 (4.1)	20.67 (8.4)

and *TB06*. The other four data sets were used at the TREC 2009-2012 Web track, and their document collection (i.e., ClueWeb09 category B) contains 50 million web pages. These data sets are denoted as *Web09*, *Web10*, *Web11*, and *Web12*.

When measuring the performance, we report the average query processing time for each data set. As discussed earlier, the query processing time of the GPU-based implementations includes the time spent on identifying top-K documents on GPU as well as the data transfer time between CPU and GPU.

5.6.1 Comparison with Exhaustive Evaluation (CPU-Based).

The most basic query processing method is to *exhaustively* evaluate all candidate documents, i.e., all documents with at least one query term for the disjunctive processing mode, and all documents with all query terms for the conjunctive processing mode. Our proposed GPU framework essentially computes the relevance scores of all the candidate documents, and can be considered as an exhaustive query processing method. Therefore, it would be interesting to compare its performance with its counterparts on CPU. The proposed GPU-based query processing methods are denoted as *GPU-OR* and *GPU-AND*. The exhaustive CPU-based top-K query processing methods are denoted as *CPU-OR* and *CPU-AND*.

Table 5.1a shows the performance comparison for the disjunctive query processing methods when K is set to 1000. It is clear that GPU-OR consistently outperforms CPU-OR methods. It indicates that processing queries in the disjunctive mode on the CPU is significantly slower than in the conjunctive mode no matter what the value of K is. However, the performance differences between the two GPU methods are very small. The latencies are almost comparable for all the values of K . This is a very encouraging finding. It has been very difficult to further improve the query processing efficiency, so researchers have started looking into how to sacrifice effectiveness, such as using conjunctive mode or document prioritization [79], to reduce the query latency. Previous study on using GPU for top-K query processing [21] proposed to optimize the efficiency by executing the conjunctive mode first and then disjunctive if there are not enough results, which indicates that there is still a performance gap between these two modes when using their GPU-based method. Interestingly, our results show that, using our proposed GPU optimization methods, we can finally bridge the efficiency gap between the disjunctive and conjunctive processing modes without making any sacrifice on the retrieval effectiveness.

5.6.2 Comparison with Dynamic Pruning Methods (CPU-Based).

Since there have been many efforts on developing more efficient query processing methods on CPU, we further compare our efforts with a few stronger baseline methods. We compare our methods with two state of the art dynamic pruning methods: maxScore [73] and Block-Max WAND (BMW) [22]. Results are summarized in Table 5.2. Clearly, the GPU-based method is much more efficient than the two baseline methods over all the data sets.

Furthermore, we conduct experiments to examine how the query processing time would be affected by K . Figure 5.4 shows the average query processing time of the GPU-OR and the two baseline methods for different values of K on the *TB05* data set. The plots on other data sets show similar trends. It is very interesting to see that the execution time of GPU-OR remains nearly the same as K gets larger, while the speed

Table 5.2: Performance comparison for disjunctive processing (ms): GPU vs. dynamic pruning (K=1,000)

	<i>TB04</i>	<i>TB05</i>	<i>TB06</i>	<i>Web09</i>	<i>Web10</i>	<i>Web11</i>	<i>Web12</i>
BMW	199.61	125.12	106.52	171.96	113.66	258.47	169.48
Maxscore	130.52	108.72	74.30	329.33	190.60	234.94	223.31
GPU-OR	21.09	17.94	16.40	31.08	21.82	43.45	31.15

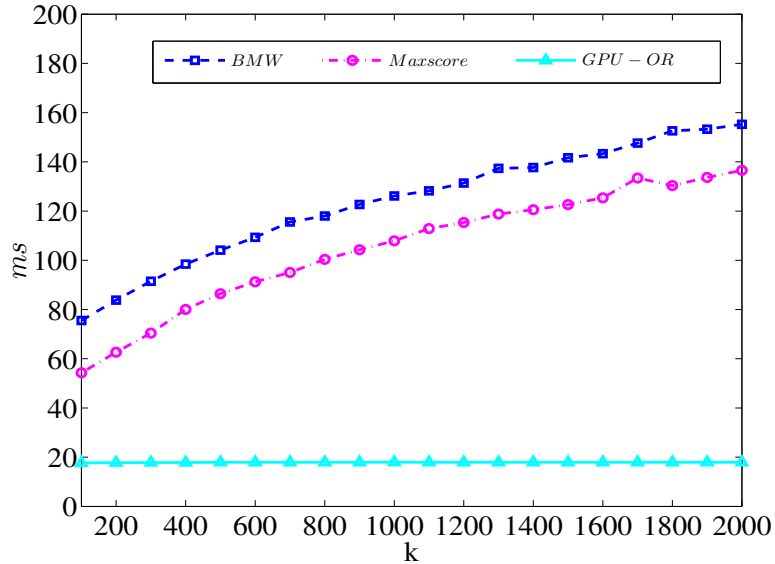


Figure 5.4: GPU-OR vs dynamic pruning (TB05)

of dynamic pruning methods increases. This observation demonstrates the scalability of the proposed GPU-based framework. It is mainly due to the final radix sorting of the framework, where the number of documents (K) to be sorted is several orders of magnitude smaller than the original document lists, accounting for only 5% percent of the total execution time. As K increases in Figure 5.4, the GPU approach stays nearly constant. A deeper analysis on the break-down of performance is presented below. On the other hand, the dynamic pruning methods evaluate more documents as K increases. Consequently, the performance difference between our GPU method and dynamic pruning ones becomes larger. The speedup scalability is a desirable property because previous studies [53, 79] suggested that a large value of K can lead to more satisfying search results.

Table 5.3: Performance comparison with the GPU baseline in disjunctive mode (ms) (K=2,000)

	<i>TB04</i>	<i>TB05</i>	<i>TB06</i>	<i>Web09</i>	<i>Web10</i>	<i>Web11</i>	<i>Web12</i>
BL-GPU-OR	95.14	94.97	96.51	95.12	93.33	100.82	95.35
GPU-OR (Speedup)	21.61 (4.5)	17.47 (5.3)	16.90 (6.0)	32.14 (3.1)	22.08 (4.3)	44.19 (2.3)	32.01 (3.1)

Table 5.4: Performance comparison with the GPU baseline in conjunctive mode (ms) (K=1,000)

	<i>TB04</i>	<i>TB05</i>	<i>TB06</i>	<i>Web09</i>	<i>Web10</i>	<i>Web11</i>	<i>Web12</i>
BL-GPU-AND	29.17	26.64	27.21	29.80	24.82	34.77	32.06
GPU-AND (Speedup)	12.79 (2.3)	11.08 (2.4)	11.42 (2.4)	23.93 (1.3)	16.2 (1.5)	25.73 (1.4)	21.73 (1.5)

5.6.3 Comparison with Previous GPU-Based Method.

We now compare the proposed GPU methods with the baseline methods proposed in the previous study [21], since this study was the first and probably the only complete solution for GPU-based top-K query processing. The authors of the previous study have kindly shared the code with us, so we directly used their codes to generate the results to ensure the correctness. Note the baseline methods assume the inverted list are stored in GPU memory and do not consider the data transfer time in the query processing time. On the contrary, our methods do not make such an assumption and the query processing time includes the data transfer time between CPU and GPU.

Table 5.3 summarizes the performance comparison for the disjunctive mode when K is set to 2000. We want to point out that, due to the different assumption made in the methods, the reported query processing time for the baseline methods (i.e., BL-GPU-OR and BL-GPU-AND) does not include the data transfer time while the reported time for our proposed method (i.e., GPU-OR and GPU-AND) includes it. As shown in the results, even when we include the data transfer time, our proposed method can still achieve an average speedup of 4 over all the collections. The results for the conjunctive mode are reported in Table 5.4. The proposed method can still outperform the baseline method.

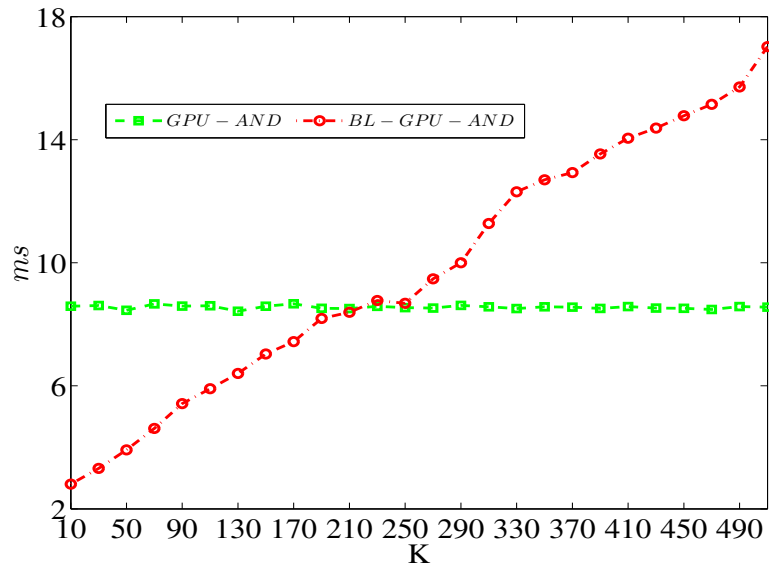


Figure 5.5: GPU-AND vs. BL-GPU-AND speed comparison of GPU and BL-GPU as K increases (TB05)

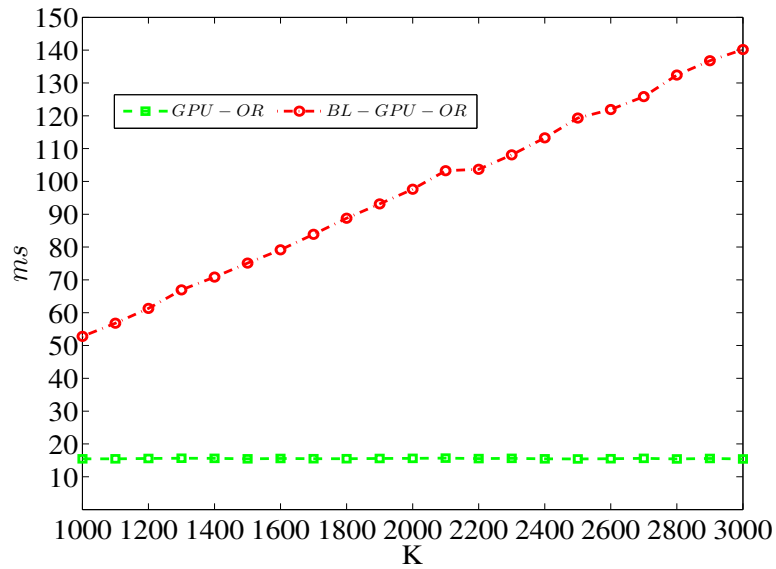


Figure 5.6: GPU-OR vs. BL-GPU-OR speed comparison of GPU and BL-GPU as K increases (TB05)

Next, we examine how the performance comparison changes with different values of K . Figure 5.6 shows the trend for the disjunctive mode. It is clear that GPU-OR method is scalable and the query processing time does not change much when we increase the value of K , while the BL-GPU-OR method does not have such a nice property. Thus, as K increases, the speedup of the GPU-OR over the BL-GPU-OR would be larger. Figure 5.5 shows the trend for the conjunctive mode. Here, we use a smaller value of K because the number of documents that contain all the query terms is not large. One interesting observation is that when K is small, GPU-AND performs worse than the baseline method. But as the value of K increases, GPU-AND becomes more efficient since the processing time of the baseline method increases linearly but the processing time of the GPU-AND does not change much.

Finally, we decompose the computations involved in the query processing time to better understand the impact of K on our methods as well as the baseline methods. In particular, we report the time spent on the three main steps: i.e., *index decompression*, *score calculation* and *top- K selection*.

Figure 5.7 shows the results for the conjunctive mode. We can see that, for the BL-GPU-AND method, the time spent on *top- K selection* increases linearly because it used maximum reduction to select top- K documents, and the overhead of looping through the maximum reduction grows almost linearly with K . On the contrary, for the GPU-AND method, the time spent on top- K selection stays nearly constant. Furthermore, BL-GPU-AND only decompressed and computed score for the intersected posting lists while GPU-AND decompressed and computed score for all the posting lists of query terms. As a result, decompression and scoring in BL-GPU-AND are more efficient than in GPU-AND. However, for the BL-GPU-AND methods, as K increases, the performance gain in the decompression and scoring steps can not compensate for its performance degradation in the top- K selection step. Therefore, the GPU-AND starts to outperform GPU-AND-BL when K increases as shown in Figure 5.5.

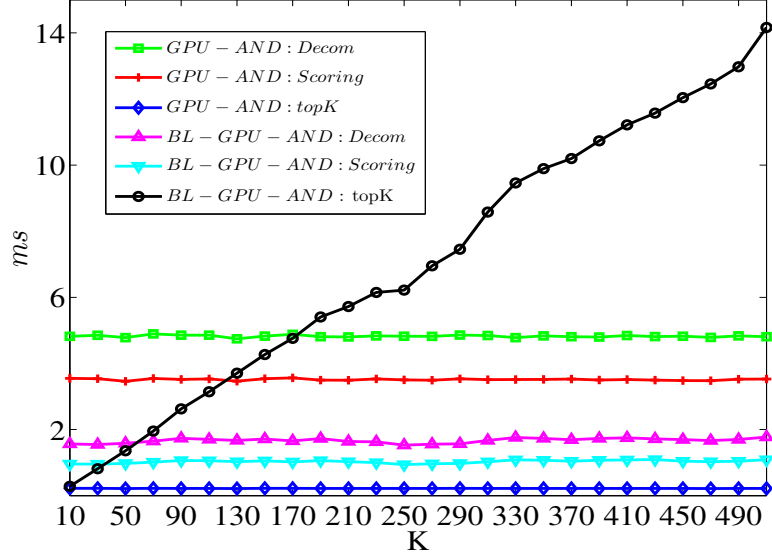


Figure 5.7: GPU-AND vs. BL-GPU-AND speed comparison of GPU and BL-GPU as K increases over three major components (TB05)

Similarly, Figure 5.8 shows the decomposed query processing time for the disjunctive mode. It is clear that the performance gap between GPU-OR and BL-GPU-OR mainly comes from the top-K selection step. BL-GPU-OR spent a significant amount of time on this step, because the maximum reduction overhead increases with the value of K . Moreover, we can see that the time spent on the decompression and scoring is about the same for GPU-OR and BL-GPU-OR. This is because when we processing queries in the disjunctive mode, the subset of the documents that the baseline method needs to consider becomes much larger.

In summary, our proposed methods demonstrate their advantages in terms of the efficiency and scalability when compared with both CPU and GPU baselines.

5.6.4 Time Analysis.

We break down the performance to understand where the speedup comes from. Particularly the data transfer time to-and-from GPU is included in our results. Figure 5.9 and Figure 5.10 show the respective percentage of the query processing time spent on each task for CPU and GPU. Note that the last step is named differently, which

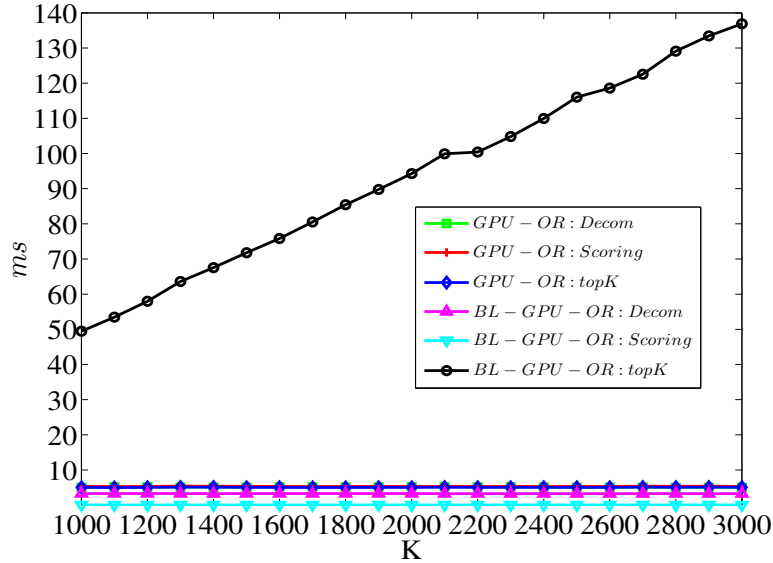


Figure 5.8: GPU-OR vs. BL-GPU-OR speed comparison of GPU and BL-GPU as K increases over three major components (TB05)

is document synchronization and top-K selection, respectively. CPU uses document pointers to synchronize among the posting lists to evaluate document in a DAAT fashion. The GPU method does not introduce such a document synchronization. Instead it deploys top-K selection to evaluate the candidate documents. For the purpose of a fair performance comparison, they should be put into the same category. We use Nvidia profiler to measure kernel running time. It is clear that each of the three steps (i.e., decompression, scoring and top-K selection) takes a big chunk of time (31%, 29%, and 27%, respectively). Beside them, the data transfer from CPU to GPU (CtoG transfer) takes the most time (12%). For each query to be executed on GPU, CPU transfers its corresponding compressed inverted index and FirstID to GPU. When GPU finishes top-K query processing, it transfers back the top-K results to the CPU. Only less than 1% is spent on GPU to CPU transfer since the size of top-K results is relatively small.

When comparing the time spent on each step by GPU and CPU, we find that the speedup of our GPU-based framework mainly comes from the score computation and the index decompression and Top-k selection gains the least speedup. The massive parallelism used in the score computation makes it possible to decrease the time spent

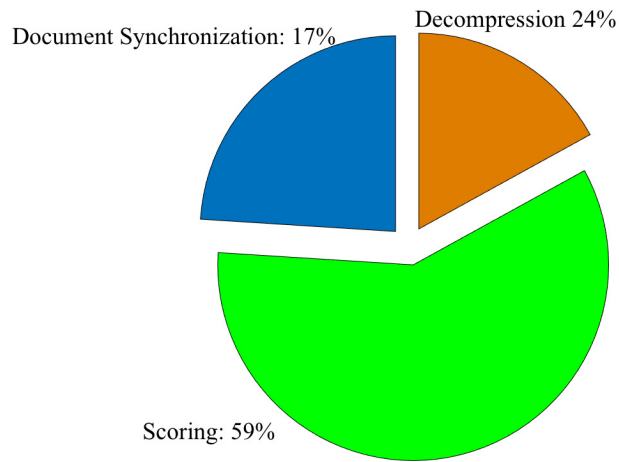


Figure 5.9: Query processing time decomposition for CPU-OR when K=1000 (TB05)

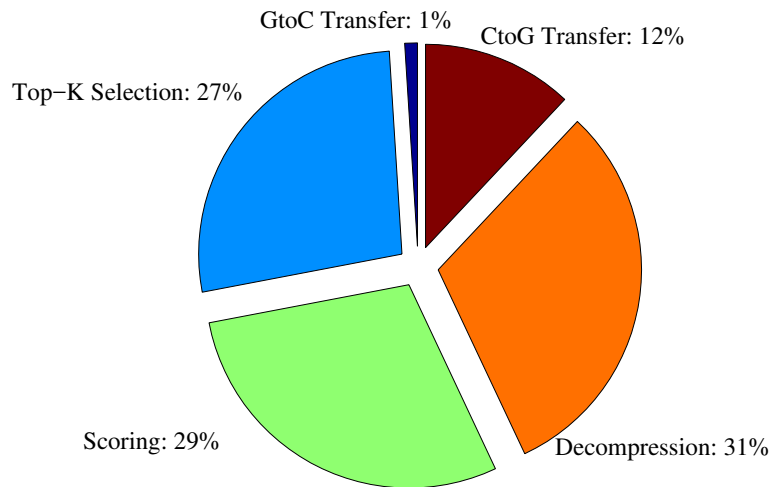


Figure 5.10: Query processing time decomposition for GPU-OR when K=1000 (TB05)

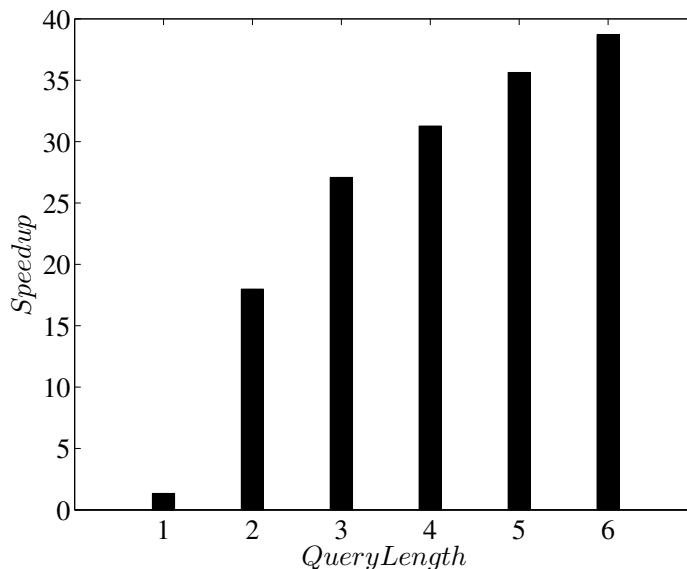


Figure 5.11: Speedup over different query lengths (*TB05*) ($K=1000$)

on computing the scores significantly. More specifically, with the CPU-based implementation, around 59% of the processing time were spent on calculating the scores. But with the GPU-based implementation, only 29% were about score calculation. In situation where massive parallelism exists, such as BM25 function in scoring step, GPU outperforms CPU significantly. Moreover, the block-based posting list is a highly regular structure with relatively high number of warp divergence and uncoalesced memory access, they lead to a loss of efficiency on GPU for decompression step. Apart from these two inefficiencies, top-K selection also suffers from atomic operations, which incur additional performance penalty on GPU.

Moreover, the break-down analysis also reveals the scalability of the GPU-based methods. As shown in Figure 5.4 and Figure 5.5, the performance of the GPU methods does not change much with the value of K . This is because the value of K only affects the *radix sorting* and *GPU to CPU transfer (GtoC)* steps. which only consists of less than 5% of query processing time. As a result, the efficiency of GPU-OR query processing methods nearly stay constant.

5.6.5 Speedup for Different Query Lengths.

One great advantage of the GPU-based implementation is the ability to process the posting lists of multiple terms in parallel, so it would be interesting to see how the speedup changes for different query lengths. Figure 5.11 shows how the speedup of GPU-OR over CPU-OR changes for queries with different lengths. It is quite encouraging to see that the speedup increases when the query length gets longer. This is a desirable property because the query processing time is closely related to the number of terms in the query. Long queries often have a long query processing time, which can cause load unbalancing and search user dissatisfaction. It is very hard to improve the efficiency of these queries without hurting the effectiveness [72]. However, our proposed GPU-based query processing framework has been shown to have great advantages in this aspect.

5.7 Summary

It is critical to improve the efficiency of Web search engines. Many CPU-based optimization strategies have been proposed for top-K query processing. Unfortunately, GPU, another powerful computational resource that is available on today's computers, has been largely under-utilized in IR systems. Our work is one of a few studies that try to bridge the gap through studying how to leverage GPU to accelerate top-K query processing.

In this work, we proposed and implemented a GPU-based top-K query processing framework for both disjunctive and conjunctive modes. We identified three important components in the framework, and discussed how to design and implement each of them by exploiting the parallel functionality provided by GPU. Empirical results over multiple TREC collections showed that the proposed GPU-based query processing methods are very efficient and highly scalable compared with both CPU and GPU baselines, in particular when the number of returned results (i.e., K) is large. Additionally, the proposed GPU-based framework can be used to achieve high efficiency

and effectiveness in search system. The implemented system and its code will be made publicly available so that others could utilize them for their own work.

This work shows that GPUs can be harnessed to accelerate top-K query processing in particular when K is large in Web search engines. There are several interesting future directions. First, there is the study of the GPU top-K execution time model to be able to predict the incoming query to GPU. Second, it would be interesting to study how to design a hybrid CPU-GPU system to co-process incoming queries based on the time model. Third, there is the possibility of exploiting the use of GPU to accelerate other components (e.g., query expansion and feedback) in a Web search engine that can potentially further improve the search efficiency. Finally, it would be interesting to study how to automatically set the parameter values in our proposed methods.

Chapter 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this dissertation, we have thoroughly analyzed how the redundancy is incurred in the im2col operation. The analysis is conducted at two levels. The discovery of the intra-row redundancy and the inter-row redundancy leads to a doubly block Hankel matrix data pattern description. This unique data pattern enables us to design and implement a new fine-grained FFT-based convolution. This dissertation presents the theoretical arithmetic complexity analysis for both our fine-grained FFT convolution and the regular FFT convolution from NVIDIA's cuDNN library. The empirical results are consistent with the theoretical analysis. This fine-grained FFT convolution outperforms the regular FFT one in terms of speed in most parts of the parameter space of the convolutions. Our efforts add to a wide spectrum of convolution approaches in CNNs, since there is no one "one-size-fits-all" convolution implementation across all the parameter space.

6.2 Future Work

There are several avenues to consider as the future work.

- Firstly, we use off-the-shelf NVIDIA's cuFFT library to implement FFT steps. cuFFT is an NVIDIA proprietary implementation of the FFT, and it is a black box library that we can not easily modify it. The library may not have the optimal performance for our limited number of power of two 1D FFT cases. To tailor for these special cases, we will implement our own FFT implementation, and it would provide performance gain over cuFFT for the sizes of interest in our fine grained FFT convolution.

- Secondly, GEMM (General Matrix Multiply) is used by all convolution algorithms except the direct convolution. Our fine grained FFT also uses a variant of GEMM. The difference is that ours is blocked matrix multiply. Within each block, it performs element-wise product. Thus it is important to optimize GEMM performance. Since the two major components of the algorithm are FFT and the matrix multiply, we are also interested in optimizing the matrix multiply operation. We tried to cast the operation as a strided batched GEMM (SGEMM) in cuBLAS library, but it showed slightly worse performance because it needs additional operations to prepare the tensors for SGEMM library calls. Although the simple autotuning strategy provides performance improvements, we plan to apply other matrix multiplication optimizations, such as register blocking, assembly code, etc., to the blocked matrix multiplication with element-wise product.
- Thirdly, if the input feature map is significantly larger than the kernel size in the regular FFT approach, too much padding occurs that could adversely affect performance and add more memory consumption. As a result, the tiling strategy can be used to mitigate the inefficiency. For example, in over-add tiling method, a $N \times N$ input feature map is decomposed into N^2/n^2 tiles that equal to the kernel size $n \times n$. It results in smaller convolution, and each smaller convolution can be computed by FFT with granularity n^2 . Compared with the normal FFT approach, the tiling strategy and our method have smaller FFT granularity, however the performance difference between them is not clear. To investigate the performance differences, we include it as our future work.
- Finally, cuDNN supports multiple convolution algorithms, and it determines the best suited algorithm under the given specification of the model based on heuristics. Since the fine-grained FFT convolution has better performance when kernel sizes and batch sizes are small, one possible future work is to develop heuristics to select the fine-grained FFT convolution when the parameters are favorable.

BIBLIOGRAPHY

- [1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.
- [2] Andrew Adinetz. Cuda pro tip: Optimized filtering with warp-aggregated atomics. *Parallel Forall. Np*, 2014.
- [3] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*, 2016.
- [4] Naiyong Ao, Fan Zhang, Di Wu, Douglas S Stones, Gang Wang, Xiaoguang Liu, Jing Liu, and Sheng Lin. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *Proceedings of the VLDB Endowment*, 4(8):470–481, 2011.
- [5] Nima Asadi and Jimmy Lin. Effectiveness/efficiency tradeoffs for candidate generation in multi-stage retrieval architectures. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*, pages 997–1000. ACM, 2013.
- [6] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The google cluster architecture. *IEEE micro*, 23(2):22–28, 2003.
- [7] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.
- [8] Andrei Z Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the twelfth international conference on Information and knowledge management*, pages 426–434. ACM, 2003.

- [9] Tom Brosch and Roger Tam. Efficient training of convolutional deep belief networks in the frequency domain for application to high-resolution 2d and 3d images. *Neural computation*, 27(1):211–227, 2015.
- [10] Chris Buckley and Alan F Lewit. Optimization of inverted vector searches. In *Proceedings of the 8th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 97–110. ACM, 1985.
- [11] Stefan Büttcher and Charles LA Clarke. Index compression is good, especially for random access. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 761–770. ACM, 2007.
- [12] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.
- [13] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.
- [14] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.
- [15] Minsik Cho and Daniel Brand. Mec: memory-efficient convolution for deep neural network. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 815–824. JMLR. org, 2017.
- [16] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. Technical report, 2011.
- [17] Jason Cong and Bingjun Xiao. Minimizing computation in convolutional neural networks. In *International conference on artificial neural networks*, pages 281–290. Springer, 2014.
- [18] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.
- [19] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.
- [20] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in neural information processing systems*, pages 1269–1277, 2014.

- [21] Shuai Ding, Jinru He, Hao Yan, and Torsten Suel. Using graphics processors for high performance ir query processing. In *Proceedings of the 18th international conference on World wide web*, pages 421–430. ACM, 2009.
- [22] Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 993–1002. ACM, 2011.
- [23] Christophe Garcia and Manolis Delakis. Convolutional face finder: A neural architecture for fast and robust face detection. *IEEE Transactions on pattern analysis and machine intelligence*, 26(11):1408–1423, 2004.
- [24] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with cuda. *IEEE micro*, 28(4):13–27, 2008.
- [25] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.
- [26] Robert M Gray et al. Toeplitz and circulant matrices: A review. *Foundations and Trends® in Communications and Information Theory*, 2(3):155–239, 2006.
- [27] Scott Gray. Maxas: Assembler for nvidia maxwell architecture, 2014.
- [28] Junli Gu, Yibing Liu, Yuan Gao, and Maohua Zhu. Opencl caffe: Accelerating and enabling a cross platform machine learning framework. In *Proceedings of the 4th International Workshop on OpenCL*, page 8. ACM, 2016.
- [29] Stefan Hadjis, Firas Abuzaid, Ce Zhang, and Christopher Ré. Caffe con troll: Shallow ideas to speed up deep learning. In *Proceedings of the Fourth Workshop on Data analytics in the Cloud*, page 2. ACM, 2015.
- [30] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.
- [31] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876, 2007.
- [32] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K Govindaraju, Qiong Luo, and Pedro V Sander. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS)*, 34(4):21, 2009.
- [33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

- [34] Tyler Highlander and Andres Rodriguez. Very efficient training of convolutional neural networks using fast fourier transform and overlap-and-add. *arXiv preprint arXiv:1601.06815*, 2016.
- [35] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2010.
- [36] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.
- [37] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.
- [38] Yangqing Jia. *Learning semantic image representations at a large scale*. PhD thesis, UC Berkeley, 2014.
- [39] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.
- [40] Heehoon Kim, Hyoungwook Nam, Wookeun Jung, and Jaejin Lee. Performance analysis of cnn frameworks for gpus. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 55–64. IEEE, 2017.
- [41] Alex Krizhevsky. cuda-convnet: High-performance c++/cuda implementation of convolutional neural networks. *Source code available at <https://github.com/akrizhevsky/cuda-convnet2> [March, 2017]*, 7, 2012.
- [42] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- [43] Andrew Lavin. maxdnn: an efficient convolution kernel for deep learning with maxwell gpus. *arXiv preprint arXiv:1501.06633*, 2015.
- [44] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021, 2016.
- [45] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.
- [46] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

- [47] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- [48] Yann LeCun, Corinna Cortes, and Christopher Burges. Mnist dataset. *URL* <http://yann.lecun.com/exdb/mnist>, 1998.
- [49] Yann LeCun, Koray Kavukcuoglu, and Clément Farabet. Convolutional networks and applications in vision. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 253–256. IEEE, 2010.
- [50] Shin-Jae Lee, Minsoo Jeon, Dongseung Kim, and Andrew Sohn. Partitioned parallel radix sort. *Journal of Parallel and Distributed Computing*, 62(4):656–668, 2002.
- [51] Xiaqing Li, Guangyan Zhang, H Howie Huang, Zhufan Wang, and Weimin Zheng. Performance analysis of gpu-based convolutional neural networks. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 67–76. IEEE, 2016.
- [52] David Lichterman. Course project for uiuc ece 498 al: Programming massively parallel processors. wen-mei hwu and david kirk, instructors, 2007.
- [53] Craig Macdonald, Rodrygo LT Santos, and Iadh Ounis. The whens and hows of learning to rank for web search. *Information Retrieval*, 16(5):584–628, 2013.
- [54] M Donald MacLaren. The art of computer programming. volume 2: Seminumerical algorithms (donald e. knuth). *SIAM Review*, 12(2):306–308, 1970.
- [55] Michael Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851*, 2013.
- [56] CUDA NVIDIA. Programming guide, cusparse, cublas, and cufft library user guides.
- [57] CUDA Nvidia. Cublas library. *NVIDIA Corporation, Santa Clara, California*, 15(27):31, 2008.
- [58] CUDA Nvidia. Nvidia cuda c programming guide. *Nvidia Corporation*, 120(18):8, 2011.
- [59] Yosuke Oyama, Akihiro Nomura, Ikuro Sato, Hiroki Nishimura, Yukimasa Tamatsu, and Satoshi Matsuoka. Predicting statistics of asynchronous sgd parameters for a large-scale distributed deep learning system on gpu supercomputers. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 66–75. IEEE, 2016.

- [60] Hyunsun Park, Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. Zero and data reuse-aware fast convolution for deep neural networks on gpu. In *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 1–10. IEEE, 2016.
- [61] Hugh Perkins. Cltorch: a hardware-agnostic backend for the torch deep neural network library, based on opencl. *arXiv preprint arXiv:1606.04884*, 2016.
- [62] Stephen E Robertson, Steve Walker, Susan Jones, Micheline M Hancock-Beaulieu, Mike Gatford, et al. Okapi at trec-3. *NIST SPECIAL PUBLICATION SP*, pages 109–109, 1995.
- [63] David Saad. Online algorithms and stochastic approximations. *Online Learning*, 5, 1998.
- [64] Eric Schurman and Jake Brutlag. Performance related changes and their user impact. In *velocity web performance and operations conference*, 2009.
- [65] Shubhabrata Sengupta, Mark Harris, Michael Garland, and John D Owens. Efficient parallel scan algorithms for many-core gpus. *Scientific Computing with Multicore and Accelerators*, pages 413–442, 2011.
- [66] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013.
- [67] Ramtin Shams, RA Kennedy, et al. Efficient histogram algorithms for nvidia cuda compatible devices. In *Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS)*, pages 418–422. Citeseer, 2007.
- [68] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [69] Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.
- [70] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.
- [71] Shirish Tatikonda, B Barla Cambazoglu, and Flavio P Junqueira. Posting list intersection on multicore architectures. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 963–972. ACM, 2011.

- [72] Nicola Tonello, Craig Macdonald, and Iadh Ounis. Efficient and effective retrieval using selective pruning. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 63–72. ACM, 2013.
- [73] Howard Turtle and James Flood. Query evaluation: strategies and optimizations. *Information Processing & Management*, 31(6):831–850, 1995.
- [74] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. Fast convolutional nets with fbfft: A gpu performance evaluation. *arXiv preprint arXiv:1412.7580*, 2014.
- [75] Kevin Vincent, Kevin Stephano, Michael Frumkin, Boris Ginsburg, and Julien Demouth. On improving the numerical stability of winograd convolutions. 2017.
- [76] Lidan Wang, Jimmy Lin, and Donald Metzler. A cascade ranking model for efficient ranked retrieval. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 105–114. ACM, 2011.
- [77] Shmuel Winograd. *Arithmetic complexity of computations*, volume 33. Siam, 1980.
- [78] Di Wu, Fan Zhang, Naiyong Ao, Gang Wang, Jing Liu, and Jing Liu. Efficient lists intersection by cpu-gpu cooperative computing. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.
- [79] Hao Wu and Hui Fang. Document prioritization for scalable query processing. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 1609–1618. ACM, 2014.
- [80] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Zhiheng Huang, Brian Guenter, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Huaming Wang, et al. An introduction to computational networks and the computational network toolkit. *Microsoft Technical Report MSR-TR-2014-112*, 2014.
- [81] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th international conference on World Wide Web*, pages 387–396. ACM, 2008.
- [82] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 59–59. IEEE, 2006.